SUN MICROSYSTEMS | WHITE PAPER

MAJC™ ARCHITECTURE TUTORIAL

TABLE OF CONTENTS

Introduction **1**

MAJC Architectural Hierarchy and Instruction Slice Architecture **2**

Features and Benefits 2

The MAJC Architecture is Scalable 2

MAJC Architecture Register Organization Drastically Reduces Wiring 4

Large Unified Register File in MAJC Achieves High Efficiency in Computing 5

Support for Saturation Arithmetic 5

Instruction Set Architecture Highlights **6**

Features and Benefits 6

Variable Length VLIW Instruction Packets 6

MAJC™ Architecture Implements SIMD and MSIMD Techniques 8

MAJC™ Architecture Incorporates an Orthogonal Instruction Set 9

MAJC does Not Mandate Hardware Interlocks at Architecture Level 9

Instruction Level Parallelism **11**

Features and Benefits 11

MAJC Implements Prediction, Control Speculation, and Explicit Predication . . . 11

Data Speculation Hides Memory Latency 13

Thread Model **15**

Features and Benefits 15

Multiprocessor-on-a-Chip Architecture **16**

Architecture Features and Benefits 16

Extremely High Coherency Bandwidth Decreases Latency and Increases Performance of Multiprocessors 16

MAJC: Extremely High Coherency Bandwidth between Multiple Processors on Chip 17

Space Time Computing (STC) **19**

Architecture Features Aiding STC and Benefits 19

How Space Time Computing Delivers Superior Performance 19

Space-Time-Computing Enhances Java Performance 21

Vertical Multithreading **23**

Architecture Features Aiding Vertical Multithreading and Benefits 23

CACHE Misses are Costly 23

Current Solutions Suffer from Deficiencies 23

Vertical Multithreading Reduces CPU Idle Time, Hides Latency and Increases Throughput 24

System on Chip/Multiprocessor System on Chip 28
Architecture Feature Aiding SOC Implementations and Benefits 28
Increased Price and Decreased Performance is a Problem in
Off-Chip Integration 28
On-chip Bus Implementations Reduce Price. Performance is Still an Issue 29
MAJC: Fast Switch Operating at Processor Speed Minimizes Latencies
and Provides Superior Performance 30

INTRODUCTION

The new services and platforms of the 21st century will place increasing demands on microprocessors. MAJC™ (Microprocessor Architecture for Java™ Computing, pronounced as "magic") is a new general-purpose microprocessor architecture from Sun Microsystems designed to address the converged services platforms and infrastructure needs of the new millennium.

Several microprocessor trends were identified and accommodated in the design of the MAJC architecture.

Processor needs for the 21st century:

- The converged services will extensively use broadband communications. The growth of bandwidth capacity on the network (approximately triples every year according to Gilder's law) is far outpacing that of processor speed (doubles every 18 months according to Moore's law). This explosive growth of bandwidth has caused information to be delivered to devices in the order of Gigabytes/second. Microprocessors should be able to process such information at "wire speed." In other words, information should be able to be processed at the rate at which it is transmitted.
- To enable "wire speed" computing and to move information in real time, extremely high bandwidth between processor units, memory and I/O devices should be available.
- The increased convergence of voice, video and data on the Internet requires general-purpose microprocessors to operate with natural data types (sight, sound and motion) and provide natural interface (voice-based) support to devices.
- There is a lot of parallelism in broadband computing. With Java™ being the platform for service infrastructure (72 percent of fortune 1000 companies use Java in 2000—80 percent on the client and 100 percent on the server), microprocessors should move beyond Instruction Level Parallelism (ILP) and address parallelism at the Thread level. Multiprocessor-on-a-chip architectures will gain significance.
- As microprocessors are used in increasingly disparate applications—from smart cards to super computers, there is a great value in the ability to create a wide span of implementations from a given microprocessor architecture. Modularity and scalability issues will be key driving forces in the design of microprocessors.
- Software, over time, will become independent of specific instruction sets; Just-In-Time (JIT) compilation techniques are expected to predominate for general-purpose processors and eliminate binary compatibility issues.
- Convergence of voice, video and data as well as price/performance issues are driving semiconductor industry toward integration of IP (Intellectual Property) blocks on a chip. Microprocessors should be designed to significantly benefit from System-on-Chip (SOC) and Multi Processor System-on-Chip (MPSOC) implementations.
- The increased use of Internet and high availability of bandwidth has caused more products and services to be delivered over the Web. Microprocessors should be able to process such information in real time and provide transparency of technology to users.
- As more businesses are conducted over the Web, microprocessors should be able to decrypt and encrypt information in real time to enable e-commerce transactions to be conducted quickly and securely over the Web.

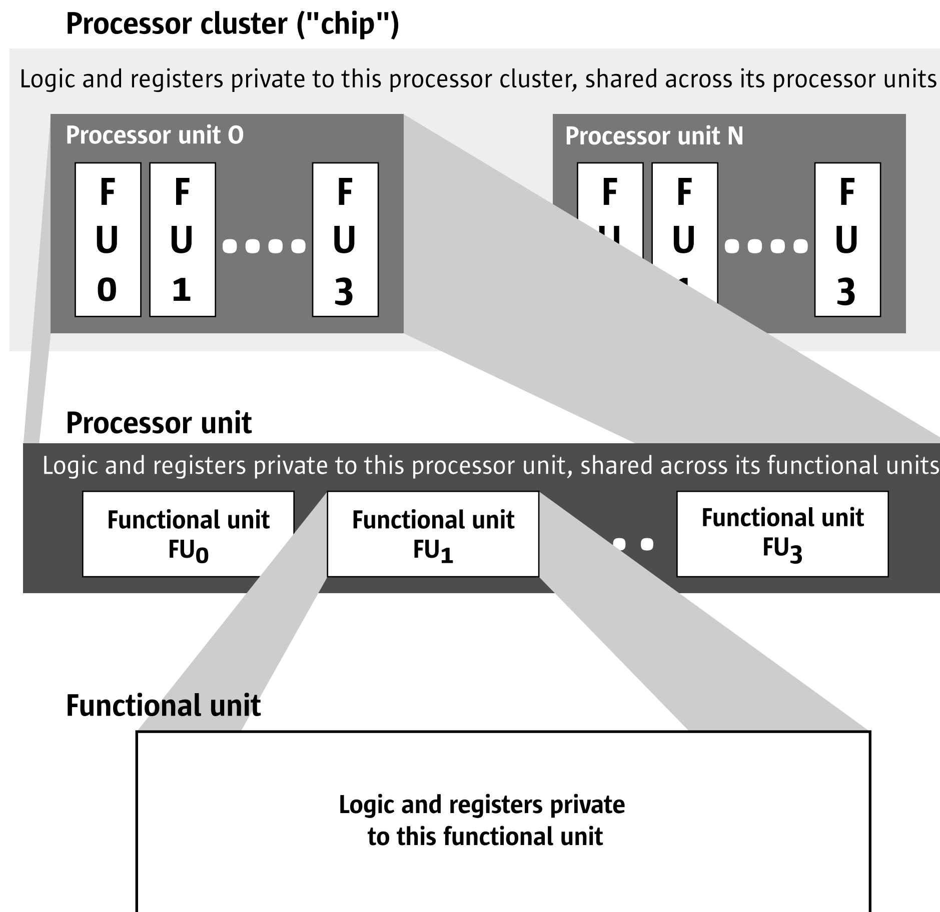
The MAJC architecture tutorial explains in detail some of the salient features of the MAJC architecture and how it is designed to address the broadband needs of the new millennium.

The tutorial is targeted at both technical and non-technical audience. However, a basic understanding of a microprocessor is essential to benefit from the tutorial.

MAJC™ ARCHITECTURAL HIERARCHY AND INSTRUCTION SLICE ARCHITECTURE

Features	Benefits
• Multiple processors can reside on a single die • Each processor is designed with its own register file, local control and state, and local wiring	• Architecture is scalable • Architecture is modular with easy to build clusters
• Multiple identical functional units can reside on an individual processor	• Maximizes functionality with minimum unique functional units
• Register organization drastically reduces wiring	• Fabrication of register files is easy • Minimizes cycle time delays
• Large unified register file	• Integers, fixed, and floating-point data coexist in all functional units • High efficiency in computation achieved

The MAJC Architecture is Scalable



The Sun MAJC architecture has three levels of hierarchy

- Processor cluster
- Processor unit
- Functional unit

Processor Cluster—Topmost in the hierarchy is the "processor cluster". A MAJC "processor cluster" contains many "processor units" on a single die. A processor is like the brain for computation and the die is analogous to a skull. The MAJC architecture, in essence, is like having one or more brains in a skull, all operating in parallel, to greatly enhance information processing capacity.

All processor units within a processor cluster are identical. The only components shared among the processor units are few internal registers (e.g. interrupts) for synchronization purposes and the memory sub-system (data cache).

Processor Unit—A MAJC processor unit can contain one to four functional units. Each of these functional units can be viewed as a RISC/DSP processor in itself. Since, in most applications, it is very difficult to find parallelism beyond four instructions and there are diminishing returns in performance beyond this, MAJC limits the number of functional units to four. Instead, the architecture extracts more parallelism at a higher level (thread level—see "Thread Model").

The functional units in MAJC are nearly identical. Functional units 2, 3, and 4 are identical, while functional unit 1 is different and can be viewed as an extended subset of the other functional units. This approach helps simplify design and implementation of processor units. It maximizes application functionality while simultaneously minimizing the number of separate functional units requiring design. Implementations typically can have 1 functional unit for the first slot and another one (repeated twice) for three other slots.

The functional unit is self contained. It has a local register file, local control (e.g. instruction/decode logic) and state information, and local wiring, not shared among processors on the same die. This makes implementing multiple processors on a die (processor cluster) very easy and customizable to the specific application.

There is a set of global registers that can be accessed by any functional unit within a processor unit. The register file size is variable and implementation specific. It can vary from as few as 32 to as high as 512. The global register file is local to the processor unit and is not shared among other processor units on the same die.

Functional Unit—Functional units are the basic building blocks of a processor unit. Individual instructions are issued to functional units. The number of instructions that can be processed in a cycle by a processor unit is dependent on the number of functional units available. Each functional unit can be thought of as a RISC processor with DSP functionalities.

Each functional unit in turn has its own private register file that is not available to other functional units. At a given time, each functional unit can work with its own local register file and the global register file.

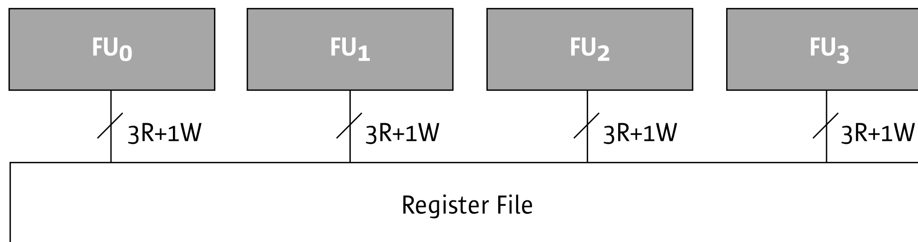
The total number of registers available for a processor unit is = number of global registers + (number of functional units x number of local registers/ functional unit)

Example: 96 global registers + 4 x 32 local registers = 224 available registers
64 global registers + 4 x 64 local registers = 320 available registers

The number of registers available to a functional unit at any point in time is = number of local registers in the functional unit + number of global registers in the processor unit. In the above example, the number of registers available to a functional unit is 128.

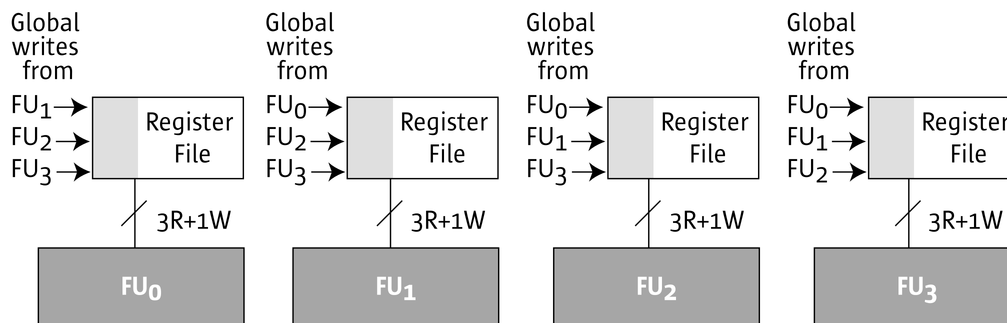
MAJC™ Register Organization Drastically Reduces Wiring

Traditional microprocessors and even recent Very Long Insturction Word (VLIW) processors have been designed where many functional units share a single global register file. This typically increases the number of register file read/write ports needed and leads to complex wiring. The increase in wiring also affects cycle time and increases latencies. For example, consider four functional units in a processor, each capable of executing instructions that involve three register reads and one write. A single register file, shared by all functional units, would require as many as sixteen ports as shown in the figure below.



Fabricating a register file with sixteen ports is very complex. The area of a register file array is roughly proportional to the square of number of ports. With sixteen ports, the area of the register file is proportional to 256, and this directly increases the register file size and affects wiring. The increased wiring in turn negatively impacts cycle time and latencies.

By contrast, the MAJC architecture reduces wiring drastically. Functional units share global registers in a processor unit and have local registers that are not available to other functional units. The register file organization in MAJC can be logically represented as follows:



From the above figure, it can be seen that each register file has three Read and four Write ports or a total of seven ports. The area of the register file array is then proportional to 4 x 72 or 196. This results in a smaller register file size, enables easier register file fabrication, and minimizes any cycle time delays.

Large Unified Register File in MAJC™ Architecture Achieves High Efficiency in Computing

Many architectures have separate register files that hold integer data, floating point data, and Single Instruction Multiple Data (SIMD). Often, applications operate predominantly on one type of data (for example, integer, floating-point, or audio/video data type). In such cases, dividing the available register resources into dedicated integer/floating point register sets limits both on register utilization and application performance. For example, in an application that requires only integer processing, register utilization will be low since the floating-point registers are unused. Performance is also affected as there are fewer integer registers available and more information saving between registers and memory is required. Further, in applications that need conversion from integer to floating point and vice-versa, data needs to be moved between register files. Such data movement between register files slows application performance.

The functional units and general purpose register files in the MAJC architecture are data type agnostic. The MAJC architecture has a large unified register file that can hold any type of data. The functional units are also data type agnostic, which means that any functional unit can work on any data type. This provides more registers for applications that involve dedicated data type processing and significantly improves performance. Further, it provides compilers with the flexibility to allocate any type of data to any register.

Summary: Any functional unit and any register file in the MAJC architecture can handle any data type resulting in high efficiency in computing.

Support for Saturation Arithmetic

The MAJC™ architecture provides saturation arithmetic to handle overflow (positive and negative) for integer, fixed-point and floating-point arithmetic. It can process 8-bit, 16-bit, 32-bit or 64-bit information. Such data types are useful for processing digital representations of naturally occurring phenomena such as sight, sound and motion. Hence, the MAJC architecture can be used to process applications involving pixels, speech data, graphics, digital communication, and audio, as well as general purpose computations.

Here are two practical examples, where saturation arithmetic is useful:

- Consider the example of controlling pixel intensities. It is well known that white is the most intense color and black is least intense color. While controlling pixel intensities, any attempt to increase pixel intensities beyond white, in a processor without saturation arithmetic, will make the pixel color black.
- Another application area of saturation arithmetic is in controlling audio volume. Suppose that audio levels can range from 1 (lowest) to 10 (highest). With a processor that does not implement saturation arithmetic, any increase in volume beyond level 10 will bring the volume level back to 1.

INSTRUCTION SET ARCHITECTURE HIGHLIGHTS

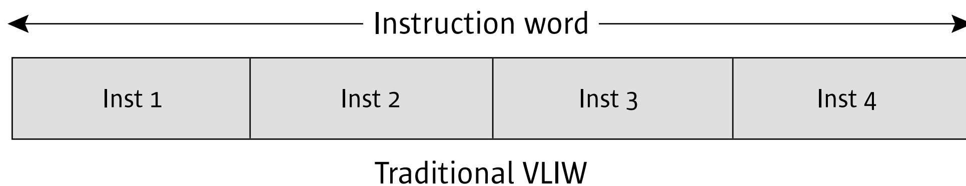
Features	Benefits
• Processes Variable length VLIW instruction packets	• Multiple instructions (1–4) can be executed in each cycle • Achieves instruction stream compression
• Incorporates SIMD and MSIMD techniques	• Applications rich in data-level parallelism are processed quickly.
• Implements an orthogonal instruction set	• Simple, yet powerful instructions can be executed quickly and efficiently.
• Does not mandate interlocking at architecture level • Incorporates scoreboarding technique	• Eliminates pipeline stalls for instructions with deterministic latencies • Optimally processes instructions with non-deterministic latencies

Variable Length VLIW Instruction Packets

EXCESSIVE CODE EXPANSION IS OFTEN A SHORTCOMING IN TRADITIONAL VLIW DESIGNS

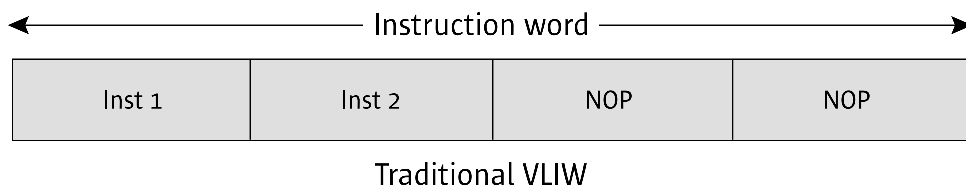
VLIW (Very Long Instruction Word) is a CPU architecture that reads a group of instructions and issues them simultaneously for execution (that is, in a single machine cycle). The compiler groups instructions and ensures that these groups of instructions are not dependent on each other and can be executed simultaneously.

In the traditional VLIW based architecture, instructions are of fixed length and are grouped into fixed long words of four to sixteen instructions. A traditional four-instruction VLIW may be represented as follows:



Traditional VLIW

The shortcoming of such a design is the "fixed instruction word". This means that if the compiler is unable to find four independent instructions, some instruction locations would have to be left empty (no operation). This leads to a lot of space being wasted and a huge code expansion. For example, a compiler that schedules only two instructions in a word will have the following representation:

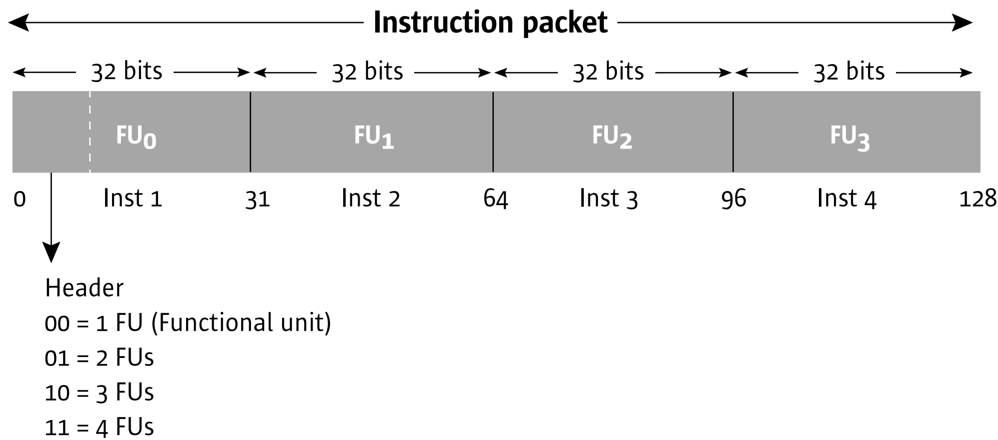


Traditional VLIW

Long fixed instruction words suffer from the problem that it is quite difficult to find more than three or four instructions to execute simultaneously in a single thread of execution. Such being the case, the code expansion will be substantial.

THE MAJC™ ARCHITECTURE ACHIEVES INSTRUCTION STREAM COMPRESSION

The MAJC architecture has a variable length (32–128 bit) VLIW instruction packet. The term "packet" is used to distinguish from a "word" as the latter connotes a fixed length. An example of a 128-bit instruction packet in MAJC can be represented as follows:

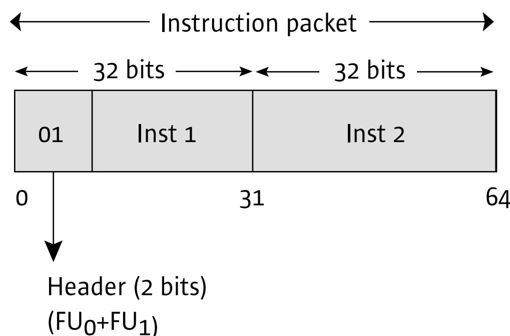


The MAJC architecture supports a maximum of four instructions in a packet, each of length 32 bits. A two bit header determines the number of instructions in the packet and, consequently, the number of functional units used by the CPU. The following header representation specifies the functional units and number of instructions processed.

Header Value	No. of Instructions in Packet	Functional Units Used	Size of Instruction Packet (bits)
00	1	FU0	32
01	2	FU0+FU1	64
10	3	FU0+FU1+FU2	96
11	4	FU0+FU1+FU2+FU3	128

A variable-length instruction packet obviates the need for NOPs to fill unused instruction slots, allowing much denser code (higher code compression). For example, a compiler that schedules only two instructions in a word (unable to find parallelism beyond this level) will have the following representation:

Instruction stream compression in MAJC



As shown above, the instruction packets will not have blanks (NOP) appended. This helps to achieve higher code compression.

MAJC instructions are issued to functional units based on their position in the instruction packet. Hence, as soon as an instruction packet is available, the functional units can "extract" instructions belonging to their respective units and begin processing in parallel.

Position in Instruction Packet	Target Functional Unit
0–31 (Inst 1)	FU0
32–64 (Inst 2)	FU1
65–96 (Inst 3)	FU2
97–128 (Inst 4)	FU3

MAJC™ Architecture Implements SIMD and MSIMD Techniques

Digital Signal Processing and multimedia applications are very rich in data-level parallelism. Large amounts of data need to be processed simultaneously. In the MAJC™ architecture, simultaneous processing of such data is achieved at three levels.

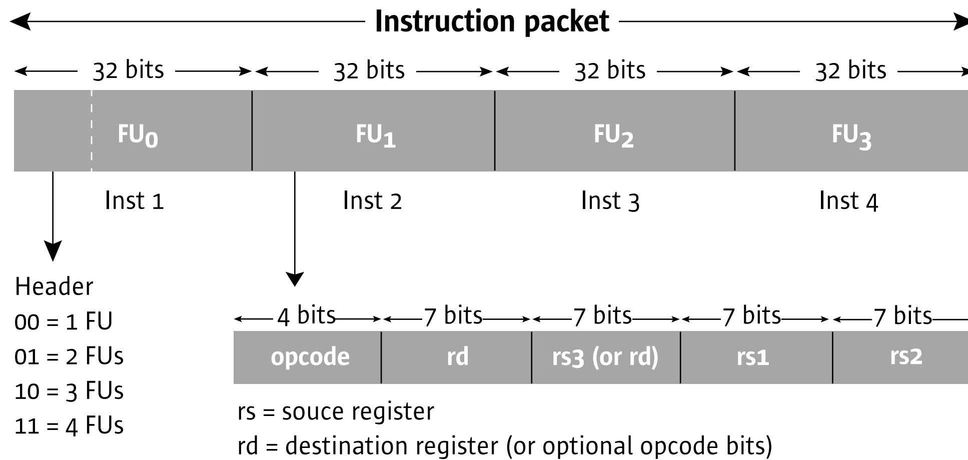
- Single Instruction Multiple Data (SIMD) DSP-like instructions in each functional unit
- Multiple Single Instruction Multiple Data (MSIMD) instructions across functional units in a CPU
- Multiple processor units per processor cluster

The MAJC architecture provides a rich set of instructions allowing multiple integer, fixed-point or floating point operations to be performed by a single instruction. This class of instructions, where a single instruction causes the same operations to be processed on multiple data, is called SIMD or "vector operation".

Let us look how SIMD and MSIMD help in real world applications. In motion estimation for video encoding, once a reference frame is transmitted, changes between the reference frame and the new frame are calculated and sent. Typically, the change between one frame and the next is not very high. A frame can be viewed as being divided into "pixel blocks". When comparing pixel blocks, one needs to compute a set of distances between pixel values, sum them up and check for any difference. In a 64-bit MAJC implementation, each functional unit can compare eight sets of pixel components (eight subtractions, eight absolute values and eight additions) using a single instruction. Further, a processor unit, comprised of four functional units, can achieve comparison of 24 sets of pixel components or 72 operations in one cycle.

MAJC™ Architecture Incorporates an Orthogonal Instruction Set

Instruction packets in the MAJC™ architecture can address upto four registers. Register specifiers are present at fixed locations, in instructions, and simplify decoding (see below).



The MAJC architecture has an orthogonal instruction set. All local and global registers in MAJC based processor units can be used interchangeably as either source or destination registers. Instruction opcodes in the MAJC architecture do not place any restriction on the type of usage of a specific register. This orthogonal feature of the instruction set enables powerful, yet simple instructions to be executed quickly and efficiently.

For example, many of the basic filtering operations used to process digitized analog signals (speech) require the mathematical operation "Multiply Accumulate," where products of many floating point multiplications must be summed up on the fly. In the MAJC architecture, three to four such Multiply-Accumulate operations can be performed per cycle per CPU!

MAJC™ Architecture does Not Mandate Hardware Interlocks at Architecture Level

Microprocessor architectures encounter pipeline stalls at data level when instructions are executed. These interlocks are either *"Register—Register"* interlocks or *"Load—Use"* interlocks.

Register—Register interlocks: Consider the following example

R8 = R6 x R7 (Multiply contents of registers 6 and 7 and store the result in register 8)
R10 = R8 + R9 (Add contents of register 8 and register 9 and store the result in register 10)

Typically, multiplication functions take about two machine cycles to execute. In the above example, if the result of multiplying R6 and R7 is not available in register R8 before executing the "add" instruction, processing of the second instruction will stall.

In order to prevent such stalls, scheduling can be performed dynamically in hardware. It requires complex hardware scheduling logic, and is often a cycle time limiter in microprocessors. Instead of devoting valuable chip area to increasingly complex hardware scheduling logic, the MAJC architecture relies on software compilers to schedule instructions.

A MAJC compiler knows the latencies in register-register instructions. For example, ADD takes one cycle, MULTIPLY takes two cycles etc. Such being the case, the compiler will consider the individual latencies when scheduling instructions (regroup instructions for execution) and ensures that the program logic is not affected.

In the above example, the MAJC compiler will know that the first instruction takes two machine cycles to execute and thus could possibly schedule instructions as follows and avoid stalls.

R8 = R6 x R7 (multiply contents of registers R6 and R7 and store the result in register R8)
R11 = R12 + R13 (add contents of register R12 and register R13 and store result in register R11)
R10 = R8 + R9 (add contents of register R8 and register R9 and store the result in register R10)

Load-Use interlocks:
Let us now look into an example of "**Load-Use**" interlocks.

Example: Load contents from memory to register R8
Compute using register R8 contents.

The latency of instructions like "load," is not predictable (i.e. how long it will take to fetch data from memory). If data is available in the cache, a "load" may take as little as two cycles. If data has to be fetched from memory, the access may take many cycles and cannot be predicted accurately.

In such cases, where instructions have non-deterministic latencies, the MAJC architecture supports *Scoreboarding* technique to monitor instructions. With scoreboarding, upon seeing a "load-use" instruction, the compiler will schedule instructions in between "load" and "use" based on the typical latencies of the instruction. The processor also maintains a scoreboard for the instruction, where it indicates whether register R8 is being loaded or is ready to be used. In our example, compiler instruction scheduling may occur as follows:

Example: Load contents from memory to register R8
Instruction 5
Instruction 6
Compute using register R8 contents

During program execution, the "load" instruction causes the scoreboard for register R8 to be set to "busy". The program goes ahead and executes instructions 5 and 6. Later, when contents in register R8 are required for further processing, the processor again checks the scoreboard. If it is still "busy", the program goes into a wait mode until R8 is populated. Else, the program uses the register R8 to process the next instruction.

INSTRUCTION LEVEL PARALLELISM

Features	Benefits
• Implements prediction, control speculation and explicit predication • Instructions in all functional units can be predicated • Special instructions implement branch predication	• Eliminates branch dependencies • Increases parallelism and decreases unpredicted penalties • Eliminates need for predicate bits
• Incorporates data speculation technique	• Hides memory latencies

MAJC™ Architecture Implements Prediction, Control Speculation and Explicit Predication

BRANCH PROCESSING IS EXPENSIVE AND LIMITS PARALLELISM

In a simple microprocessor, instructions are typically processed in four stages: fetch, decode, execute and write. We can save instruction processing time if, for example, we can decode a new instruction while the previous instruction is executed or fetch a new instruction while decoding the previous instruction. This process of overlapping instruction execution is pipelining and is similar to an assembly-line process. The potential overlap of instructions either in a pipeline (overlap in time) or across functional units of a processor (overlap in space) is called Instruction Level Parallelism (ILP).

Lengthening the pipeline by including more stages in the pipeline, to overlap many instructions and increase parallelism, produces diminishing (or even negative) returns in normal code. This is due to the dependencies in the pipeline which can stall the process. Dependencies can be due to data (result of an instruction is required for processing the next instruction), control (change in program flow due to branches) and name (instructions using same register space or memory location).

A branch (control dependency) is the result of an IF(condition)-THEN-ELSE construct. If the condition is true, the “THEN” instructions are executed. If the condition fails, the “ELSE” instructions are executed.

In pipelined processors, conditional branches are encountered before the data that determines the branch direction. As instructions are fetched ahead of their execution, correctly predicting control flow is critical. If the branch flow is incorrectly determined (mispredicted), the pipeline must be flushed off the incorrect instructions and stalled until correct instructions are fetched. In heavily pipelined architectures, mispredicted branches reduce performance.

PREDICTIONS REDUCES BRANCH DEPENDENCIES. MISPREDICTIONS ARE COSTLY

Branch mispredictions can be reduced by good hardware/software prediction. In simple terms, the microprocessor predicts branch outcomes based on previous results. That is, it assumes that the past predicts the future. If a particular branch executed the THEN condition 90 percent of the time in the past, the chances of the branch executing the THEN condition again is high. The processor then “speculatively” fetches the first instruction in the THEN condition. If however, that prediction turns out to be wrong, then the pipeline needs to be cleared and the correct instruction from the ELSE condition needs to be fetched.

Prediction is similar to the processing at some of your favorite Pizza centers. Suppose you enter the Pizza center of your choice and typically order a mushroom pizza. The person taking the order, upon seeing you in the line, can “speculatively” order a mushroom pizza for you (based on your past business) and effect fast service (and probably win your loyalty!) when you order the same pizza. If, on the other hand, you wanted a cheese pizza, then the mushroom pizza needs to be withdrawn from the pizza production line and the order taker (“order fetcher”) needs to place a new order for cheese pizza. This causes a delay and affects service.

The MAJC architecture supports both static and dynamic prediction. In static prediction, the MAJC compiler directly tells the hardware whether a branch is expected to be taken or not. This is called branch steering. In dynamic prediction, the hardware predicts branch directions by using a run-time technique, such as a branch history table. Static prediction techniques, such as branch filtering and branch steering, help to control the entries in the branch history table. The limited size of the branch history table will contain entries of only those branches which are difficult to predict at compile time. This avoids branch history table "pollution" and results in higher aggregate branch prediction.

Prediction is very useful when branch behaviors can be more or less accurately predicted. In some cases, the behavior of a branch is not well known and, as such, cannot be predicted with a high degree of accuracy. In such cases, mispredicts can be expensive—especially in heavily pipelined architectures.

CONTROL SPECULATION AND EXPLICIT PREDICATION ELIMINATE BRANCH DEPENDENCY

Returning to our pizza example. Suppose that pizza ingredient costs nothing and there was an extra oven available. So, before you order, the person taking the order can now go ahead and prepare both mushroom and cheese pizzas for you. He can then allow you to "pick" the pizza of your choice on that day and discard the one you did not order, thereby reducing any delays and improving service. This in simple terms is the basis for control speculation with predication.

Predication refers to the conditional instructions that can be used to eliminate branches completely. A single instruction can be used to execute the condition. This assists the compiler in moving instructions past the branches.

Speculation refers to executing a code segment before it is known whether the results of that code segment will be used or not. To illustrate this, let us consider a simple example.

```
If (flag = '0')
  OUT <— A
Else
  OUT <— B
Endif
```

In a traditional architecture, the pipeline needs to wait and determine the contents of the variable "flag" and then move either "A" or "B" to variable OUT. The processing of the condition stalls the pipeline and limits the parallelism that can be exploited.

However, using control speculation and predication, such stalls can be reduced. There are a few ways to implement this, all of which are equally effective. One way to implement predication is to associate each and every instruction with predicate bits and to have predicate registers determine either the "True" or "False" conditions. This is similar to tagging instructions with a flag called "color flag" (predicate bits). The "color flag" may be set to either red or green to identify results as belonging to the THEN statement or ELSE statement. Upon executing the condition, the predicate register is set to either "Red" or "Green" color. Depending on the "color" in the predicate register, either instructions from THEN or ELSE statements are used.

Predicate register specifiers within instructions are expensive. It reduces instruction density—that is, the maximum number of instructions that can be executed in a given packet size. The MAJC architecture eliminates the need for predicate bits in every instruction. Instead, it uses a specific set of predicated instructions. Each MAJC CPU has four functional units, all of which can execute predicated instructions. This means that in a single cycle, each processor unit can execute upto four predicated instructions, all operating on an orthogonal register file. For example, the Move Conditional instruction copies one register to another if and only if a third register, the "condition register," is zero (or non zero). Other examples

are Pick Conditional (based on contents of the first source register, it copies the value of one of two other registers to the destination register), Move Conditional Parallel, Store Conditional and so on. These instructions increase execution efficiency by eliminating many conditional branches and are as effective as burdening all instructions with the predicate bits.

The MAJC architecture supports control speculation with predication. The support for speculation in MAJC enables multiple paths of IF-ELSE conditions to be executed in parallel, increasing the parallelism that we can exploit. In the end, predicated instructions (such as pick conditional) can be used to select which results are to be used from the multiple code-segments that were executed. For example:

```
If (condition) THEN
 ......
 ...
 ....
ELSE
 ......
 ....
 ....
ENDIF
```

In the above case, Functional Unit 1 may be working speculatively on the assumption that condition = TRUE and simultaneously Functional Unit 2 on the assumption that condition = FALSE . In the end, "Pick Conditional" can be used to select the correct result.

Using control speculation with predication, the example in the earlier part of the section can be implemented as:

```
Register R8 <— Flag
Register R9 <— A
Register R10 <— B
Register R11 <— OUT
PICKC R8, R10, R9, R11
```

The last instruction says: check for contents of register R8. If the value is "1," copy contents of register R10 to R11. If the value is "0," copy contents of register R9 to R11.

Data Speculation Hides Memory Latency

Loads from memory are always time consuming relative to on-chip operations, so this memory latency affects performance. Stores to memory, on the other hand, are not as critical an issue as there are efficient ways to overcome store-latency.

MAJC™ architecture permits software to "speculatively load" the contents of memory to a register before it is certain that the results of the load will be used. This means that data is loaded independent of the condition. Here is a simple example:

```
If(condition)
OUT <— 1st element of array
Else
OUT <— 2nd element of array
EndIf
```

In a traditional architecture, the processor loads the first element of an array to variable OUT only after validating the IF condition. The loading of the array element is time consuming and this delay can be avoided. If we move the load outside of the branch (after certain checks to eliminate process flow conflicts), we can obtain an optimum instruction scheduling, which may look as follows:

```
OUT1 <— 1st element
OUT2 <— 2nd element of array
  ......
  ...
  ....
If(condition)
OUT <— OUT1
Else
OUT <— OUT2
EndIf
```

Given the latencies in the memory system, the contents of OUT1 and OUT2 will be available by the time the condition executes. A possible implementation of the same on the MAJC architecture might read as follows:

```
Register R8 <— condition
Register R9 <— 1st element of array
Register R10 <— 2nd element of array
Register R11 <— OUT
 ......
 ...
 ....
PICKC R8, R9, R10, R11
```

The last instruction says: check for contents in register R8, if the condition is true, copy the contents of register R9 to register 11, else copy the contents of register R10 to register R11.

Speculative loading in the MAJC architecture is non-faulting. This means that if there is any error while loading the first (or second) element of the array into OUT1 (or OUT2—for example, misaligned memory address, data access error etc.), the value of 0 is stored in the destination, instead of a value from the memory. Later in the program, if the contents of OUT1 (or OUT2)=0, the processor executes the load again and finally copies the contents of OUT1 (or OUT2) to OUT.

Hence, we see that speculative loading (non-faulting loads) helps hide memory latency. This enables the processor to extract more parallelism and improve performance. The MAJC architecture also permits for explicitly prefetching data ahead of time to decrease memory latency.

In summary, the MAJC architecture supports speculation in the following ways:

- *Each processor has multiple functional units that can do computations in parallel*
- *All functional units of the processor have predicated instructions*
- *Non-faulting loads enable speculative loading*
- *Compiler supports speculation*

THREAD MODEL

In today's applications, even with aggressive predication and speculation, it is difficult to consistently find more than four independent instructions that can be simultaneously executed. Hence, investing chip space in high order issue width often results in diminishing performance returns.

The next few chapters provide more details about the "Thread model" in the MAJC™ architecture, where parallelism is exploited at the thread level and application performance is substantially increased. Certain new techniques that exploit thread-level parallelism are also explained.

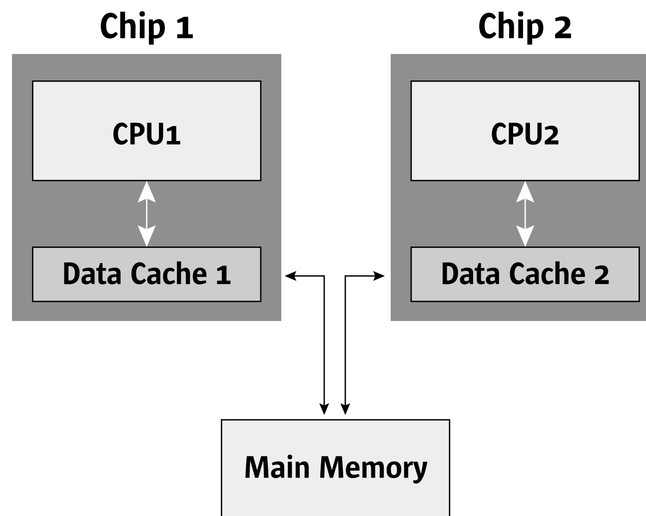
Architecture Features	Benefits
• Multiple processors can reside on a chip	• Exploits instruction- and thread-level parallelism • Independent threads, applications, processes can execute in parallel
• Supports Space Time Computing technique	• Speculative threads executing across processor units substantially improves performance of many single-threaded and multithreaded applications.
• Supports Vertical Multithreading technique	• Multiple threads executing within a processor unit, upon a cache miss, significantly reduce CPU idle time and increases throughput

MULTIPROCESSOR-ON-A-CHIP ARCHITECTURE

Architecture Features	Benefits
• Multiple processor units can reside on the same die	• Processes and threads can execute simultaneously and efficiently
• Processor units can either: - Share a first level data cache AND/OR - Run on a very low latency on-chip coherency bus operating at processor speed	• Extremely high coherence bandwidth between processor units is achieved and superior parallelism at processor level is attained • Fast interprocessor communication • Processes and threads execute free from conflicts and jointly perform tasks rapidly.

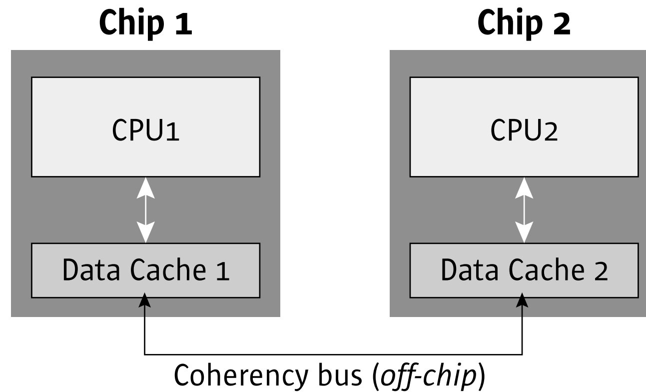
Extremely High Coherency Bandwidth Decreases Latency and Increases Performance of Multiprocessors

A "cache-coherence problem" refers to a condition in which two or more processors in a shared-memory multiprocessor system observe different values (in their caches) for the same memory location. Current architectures employing multiple processors are unable to achieve significant improvement in performance due to issues with inter-processor thread synchronization and communication. Following are descriptions of some methods in which cache coherence synchronizations between multiple processors are achieved in various system designs.



In the above design, the CPUs have dedicated data caches. Data communication between the CPUs is achieved by writing the contents of the data cache to the main memory and then loading contents of the main memory to the other data cache. However, communicating information between data caches through main memory is time consuming, often in the order of hundreds of processor cycles. This causes cache-coherence problems.

A second design involves communicating between data caches using an off-chip coherency bus as shown below.



Although this design is an improvement over the previous memory-based communication, it still has potential coherency bandwidth bottlenecks. Communication between data caches is a function of the speed of the off-chip coherency bus, which is typically between 66–200 MHz, and the bus width. For processors operating at say 500–600 MHz, the latencies incurred on such an off-chip coherency bus are not negligible.

MAJC™ Architecture: Extremely High Coherency Bandwidth Between Multiple Processors On Chip

MAJC architecture allows parallelism to be exploited at the system level through its intrinsic support for multiple processors on a chip. The architecture permits more than one processor unit to reside on a single chip (called a processor cluster). Processor units in MAJC implementations can either share a first level data cache or run on a very low latency on-chip coherency bus, operating at processor speed. This allows MAJC implementations to achieve extremely high coherent bandwidth between processor units. The figure below shows possible MAJC implementations of multiple processor units on a chip.

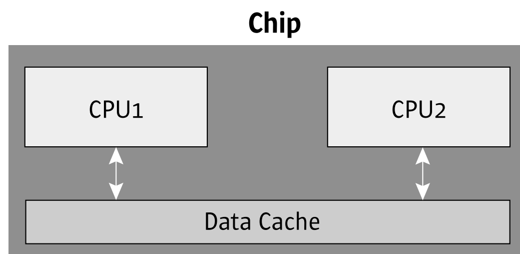


Figure a

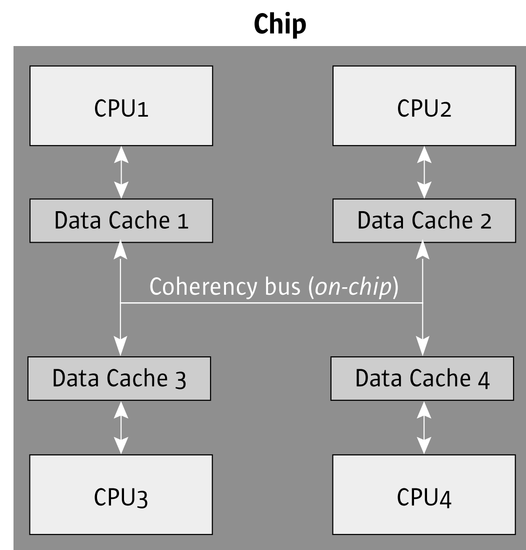


Figure b

Figure (a) shows a possible MAJC implementation employing a "join at the hip" mechanism, where two processor units, sharing a common data cache reside on a single chip. Shared data is available to the processors in real time: as soon as one processor writes content to the data cache, the other processors can read that content from the data cache immediately. Processors can also simultaneously read the same data location, read and write different data locations or exchange locks in few cycles as opposed to hundreds of cycles in traditional multiprocessor systems.

Figure (b) shows another possible MAJC implementation where multiple MAJC processor units run on an on-chip coherency bus. The coherency bus runs at processor speed and achieves fast interprocessor communication with very low latency (few cycles).

The MAJC architecture is designed to limit complexity of the processor units in terms of issue-width (degree of ILP), while attaining superior performance at the thread level. In today's applications, even with aggressive predication and speculation, it is often hard to consistently find parallelism beyond four instructions per cycle. The MAJC architecture enables parallelism to be exploited at the thread level instead of wasting functional units by just extending the issue-width. This feature is particularly interesting for applications using Java™ technology, which commonly have multiple threads of execution.

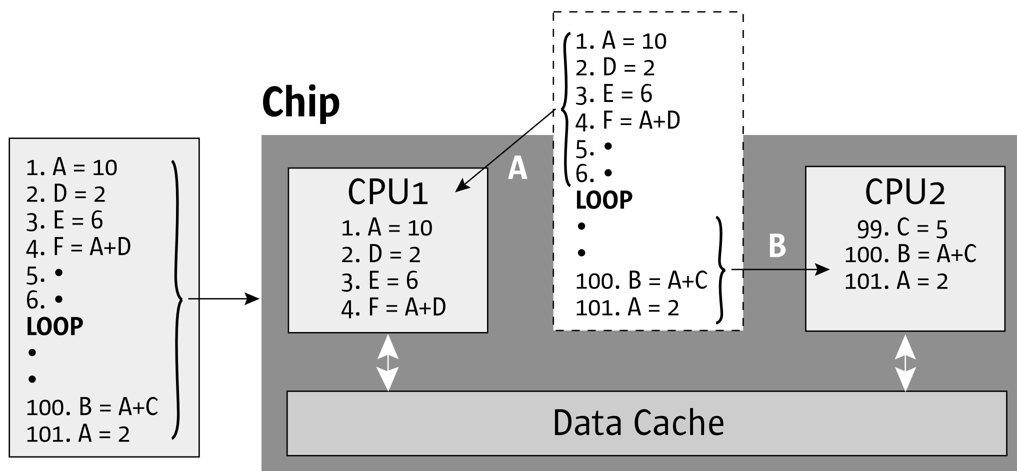
Applications that mainly exhibit single-threaded profile (that is, either have very few threads or have multiple sequential threads) are sped through vertical multithreading techniques (Chapter 6: Vertical Multithreading).

These features in the MAJC architecture enable processors support an execution environment where independent processes, independent threads, or tightly coupled threads sharing the same datastructure can run simultaneously. For example, when one Java virtual machine (JVM) is running on a processor, parallel threads can belong to the same application, different applications, to the operating system, or to the runtime environment itself.

SPACE TIME COMPUTING (STC)

Space Time Computing in the MAJC™ architecture is a technique that substantially improves performance and code execution time in many applications using Java Technology. The multiprocessor-on-a-chip configuration in the MAJC architecture allows system level parallelism on a processor cluster to be achieved by having speculative threads (future instruction streams) execute on separate processors. For example, if we have two processors on a chip, then two threads—Head and Speculative—execute on separate processors. They operate in a different space (speculative heap) and in a different time (future time).

Architecture Features Aiding STC	Benefits
• Architecture supports multiple processor units on a chip • Shared first level data cache (or a very low latency on-chip coherency bus) between processor units offer extremely high coherence bandwidth • Relaxed memory order allows load/store instructions to be executed out of program order, while special instructions ensure that no extra atomic (memory barrier) instructions are needed to ensure correct outcome	• Substantially improves performance and code execution time of many Java applications • Threads can be synchronized effectively
Java™ Technology Aiding STC	
• The Java programming language is a structured programming language with Object Oriented properties • Only stores to heap result in a rollback • Pass by value semantic for primitive data types does not affect heaps. Only object types affect heaps • Stores to heap for object types in Java can be accurately determined at the byte-code level	• Rollbacks can be determined accurately • Minimal overhead to the head thread • Additional computations of the speculative threads overlap with JVM activities and effectively hides most of the overheads



A = Head thread
B = Speculative thread
Note: Threads operate in their own space.
Detects Read after Write, Write after Write, and Write after Read violations.

Note: This example is not related to Java (or any object oriented language), but is a simple code to explain Space Time Computing.

How Space Time Computing Delivers Superior Performance

To better understand Space Time Computing, consider a simple program executing on two processors. When programs have loops or method boundaries (as with Java), the MAJC architecture splits the program into instruction groups (threads) that are executed simultaneously on different processors as shown in the figure above. The first set of instructions or instruction group runs on CPU1 and is called the "head thread" (A). The second instruction group executed on the CPU2 is called "speculative thread" (B). It is termed speculative since it is executing the instruction group ahead of time. Thus program instructions that would be executed in "future time" have been made "current."

When program groups or methods are executed ahead of time, two aspects need careful attention:

- Speculative threads should not modify the processing or computations in the head thread.
- Speculative threads should always work with the coherent program order values.

In our example, we need to ensure that changes to variables in the speculative thread on CPU2 (future time) do not affect computations in the head thread in CPU1 (that is, when computing instruction 4, CPU1 should use value of A=10 and not A=2 from CPU2). The MAJC architecture allows speculative threads to operate in their own space called the "speculative heap", where modification to common program variables in CPU2, which are also used in CPU1 (value of A=2 in our example), will be local to speculative thread and will not be reflected in the head thread. Operating with speculative heap also ensures that any processing of variables at the end of STC (after joining the head and speculative threads) will use the latest results as written by CPU2 in the speculative heap.

On the other hand, it is possible that changes to variables made in the head thread are used in the speculative threads. When the head thread and speculative threads are executing simultaneously, it is important to ensure coherence (the speculative threads should use the correct program order value of variable). For example, it is possible that instruction 5 in CPU1 is a store instruction, that writes variable A with the value 5 (instruction 5: A=5). If there are no more changes to variable A in instruction group (head thread) A, the speculative thread on CPU2 should use the value of A=5 in computing the value for B (as against 10 at the start). When there are such conflicts, the MAJC architecture issues a "rollback" and the speculative thread B starts executing again.

IN REAL WORLD APPLICATIONS, SUCH ROLLBACKS ARE RARE. THERE ARE FEWER STORES COMPARED TO LOADS ...

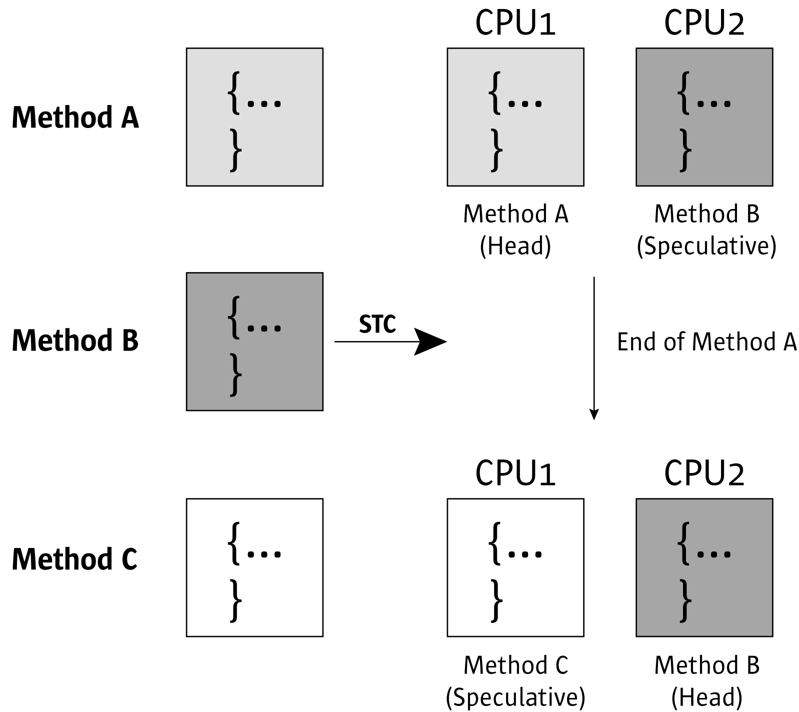
The discussions in the last couple of paragraphs can be summarized in the following table:

Hazards in Programs Created by Speculative Execution	STC Solution
• Read After Write (RAW): CPU1 writes data. CPU2 reads contents of the same data. Result expected: CPU2 should use the latest write of CPU1	• Rollback: Restart execution in CPU2 when CPU1 modifies contents read by CPU2
• Write After Write (WAW): CPU1 writes data, CPU2 modifies the same data. Result expected: After join of head and speculative threads, CPU2 write should be reflected in the end result	• Speculative heap: Writing in the speculative heap by CPU2 ensures correct outcome at the end of STC
• Write After Read (WAR): CPU2 writes data, CPU1 reads data. Result expected: CPU1 should NOT read results written by CPU2, but should use values modified by CPU1	• Speculative heap: Writing in speculative heap by CPU2 ensures that CPU1 does not use results of CPU2

Space Time Computing Enhances Java™ Programming Language Performance

Space Time Computing is extremely useful when we are dealing with Structured Program Languages such as the Java programming language. The following descriptions substantiate why applications based on Java technology benefit from Space Time Computing.

Let us look at a Java technology-based program that invokes three methods A,B, and C which run on a two CPU implementation of the MAJC architecture.



CALLING SEMANTICS OF JAVA HELPS STC—STORES TO STACK SEMANTIC FOR PRIMITIVE DATA TYPES AND STORES TO HEAP SEMANTIC FOR ALL OBJECT TYPES

At the start of the program, method A will execute on CPU1 as the head thread and method B will execute on CPU2 as the speculative thread. When the program invokes method B, all parameters required for method B (data types integer, float, double etc.) are "passed by value." The only stack value of method A that could be changed at the completion of method B is the return value. Nothing else in method A will be affected. Hence, stack values after executing a void method are known at the start of the next method. Typically, more than 60 percent of all functions do not return any value of importance. They either return void or return a value that is not used.

Stores to heap are of the greatest importance. Heap is an area in memory where object types such as structures, files or in general objects whose memory is determined at execution time are stored. Different methods can access heaps and modify its contents. For example, when method A and B are executing, method A can modify the contents of the heap. If method B is already working on the previous copy of the heap contents, then we need to rollback and start executing method B again.

A ROLLBACK IS REQUIRED ONLY ON STORES TO HEAP

As we have seen earlier, stores to stack do not affect program order. For stores to heap, the MAJC™ architecture is designed to effectively synchronize the threads in cases of rollback. When method A and method B are executing in parallel, any store to heap in method A is monitored by the head thread. CPU1 (head thread) checks to find if CPU2 (speculative thread) is already working with the old copy of the heap contents. Such being the case, the head thread issues an interrupt (enabled by fast cross calls between processors) to CPU2 to rollback and start method B again. However, if method B wants to modify the heap contents (as speculative thread), it will "create a copy" of the required heap contents locally, ensuring that the head thread is not affected. It is important to note that the flow of the head thread is not affected in any major way.

At the end of executing method A, method B (executing on CPU2) will become the head thread and method C (executing on CPU1) will become the speculative thread. The process continues.

JAVA™ TECHNOLOGY HELPS DIFFERENTIATE BETWEEN STORES TO HEAP AND STORES TO STACK AT THE BYTECODE LEVEL

In the Java programming language, one can differentiate between stores to heap and stores to stack at the bytecode level. This is a big advantage for Space Time Computing. It helps compilers in issuing special instructions that help to accurately determine possible rollbacks. Thus, fewer stores are needed to be monitored.

In summary Java technology significantly benefits from Space Time Computing:

- *Rollbacks are rare as there are fewer stores compared to loads*
- *Not all stores are important. Only stores to heap should be monitored*
- *Stores to heap can be monitored at the bytecode level*
- *Rollbacks can be accurately determined*

VERTICAL MULTITHREADING

Vertical Multithreading is a technique where multiple threads operate on a cache miss within a processor unit and reduce the overall CPU idle time. It decreases the total CPU cycles required for program execution and increases throughput.

Architecture Features Aiding Vertical Multithreading	Benefits
• Multiple threads can operate in a processor unit on a cache miss • Large unified register file maintains multiple thread references • Register space for threads can be monitored and controlled	• Vertical Multithreading significantly reduces CPU idle time and increases throughput

CACHE Misses are Costly

A cache is a small, fast memory, located close to the CPU that holds the most recently accessed code or data. When the CPU finds requested content (data/instruction) in the cache, it is called a "cache hit." When the CPU does not find the content in cache, it is called "cache miss."

In the case of a cache miss, the CPU needs to load the content from the main memory to the cache. The typical wait time for a CPU, before it resumes processing, is between fifty to one hundred cycles. Access times can even be longer if the processor must contend with other devices for accessing memory. It has been observed in commercial and service provider applications that there is a five percent cache miss for every one hundred instructions.

For example, suppose we have one hundred instructions in a program, each taking one cycle

Begin Program
inst 1
inst 2
inst 3
.
.
inst 100
End program

Given the five percent cache miss rate, we see that there will be 95 cache hits and five cache misses. The 95 cache hits will take 95 CPU cycles to complete. The five cache misses will result in the CPU idle for 375 cycles (approximate wait time for a cache miss is 75 cycles). In other words, the CPU will be utilized for 95/(95 + 375) or about 20 percent, while it will be idle for about 80 percent of the time.

Current Solutions Suffer from Deficiencies

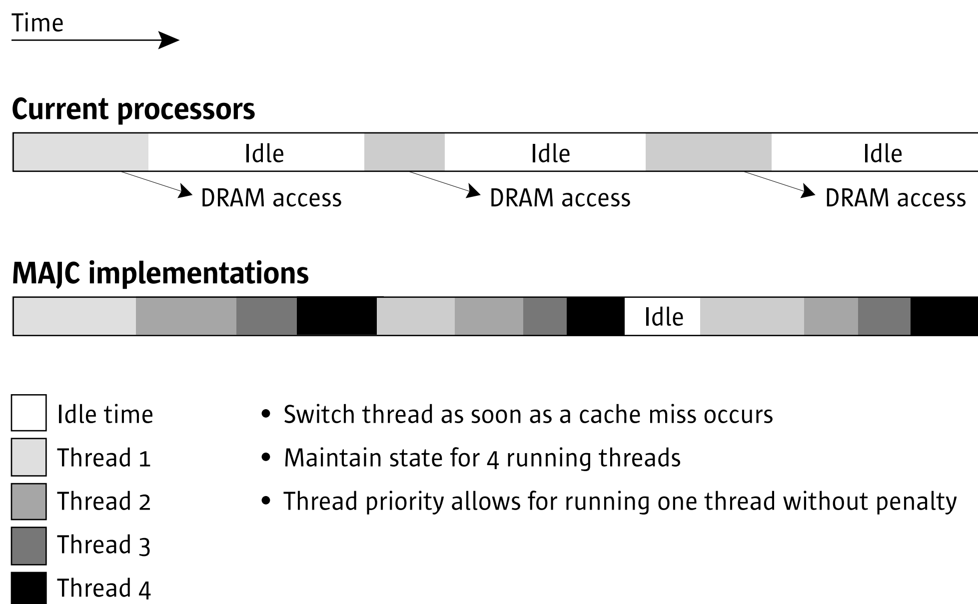
The most common ways to reduce CPU idle time are to use either large caches or compiler prefetches. Large caches are useful when data is needed repeatedly as in the case when data exhibits either spatial or temporal behavior. Prefetch and streaming non-cachable data (both supported by MAJC) are well suited for many multimedia applications where data streams "IN" and "OUT."

However, in many commercial applications "cold requests" are made in a somewhat random pattern (non-repeatable) to the memory subsystem. In such cases, having a large cache or deploying sophisticated prefetch techniques does not alleviate the problem.

Vertical Multithreading Reduces CPU Idle Time, Hides Latency, and Increases Throughput

The MAJC™ architecture substantially increases CPU utilization with vertical multithreading. A thread is a stream of instructions. In the simplest terms, vertical multithreading in the MAJC architecture allows the CPU to switch to a new instruction stream whenever there is a cache miss. This increase in CPU utilization, coupled with instruction level parallelism gained from VLIW, significantly improves overall throughput.

In order for vertical multithreading to be effective, the load time from the memory to the cache (upon cache miss) should be much greater than the switch time between instruction streams. To enable fast context switching, references to the threads (states) need to be stored. The large register file in MAJC allows the architecture to maintain reference information of four threads and effect fast switching between instruction streams. The MAJC architecture permits monitoring and trapping any register overlaps that may occur when storing reference information of multiple threads. This is important to ensure accuracy in processing thread information.



Having seen the basic idea of vertical multithreading, let us see how the MAJC architecture brings about performance improvements over traditional architectures.

Here is a simple example that will bring out a couple of key benefits of vertical multithreading. Consider a program with 100 instructions, having four cache misses during execution, one each after every twenty instructions. Let the approximate wait time for memory contents to be loaded to cache be 75 cycles. Further, let the average number of program instructions that can be executed in a single four-issue wide VLIW instruction packet be 2.5. In other words, we are assuming that some instruction packets execute four program instructions per cycle, while still others may execute 3, 2, 1, or 0 program instructions per cycle. The average number of program instructions executed in each cycle is 2.5.

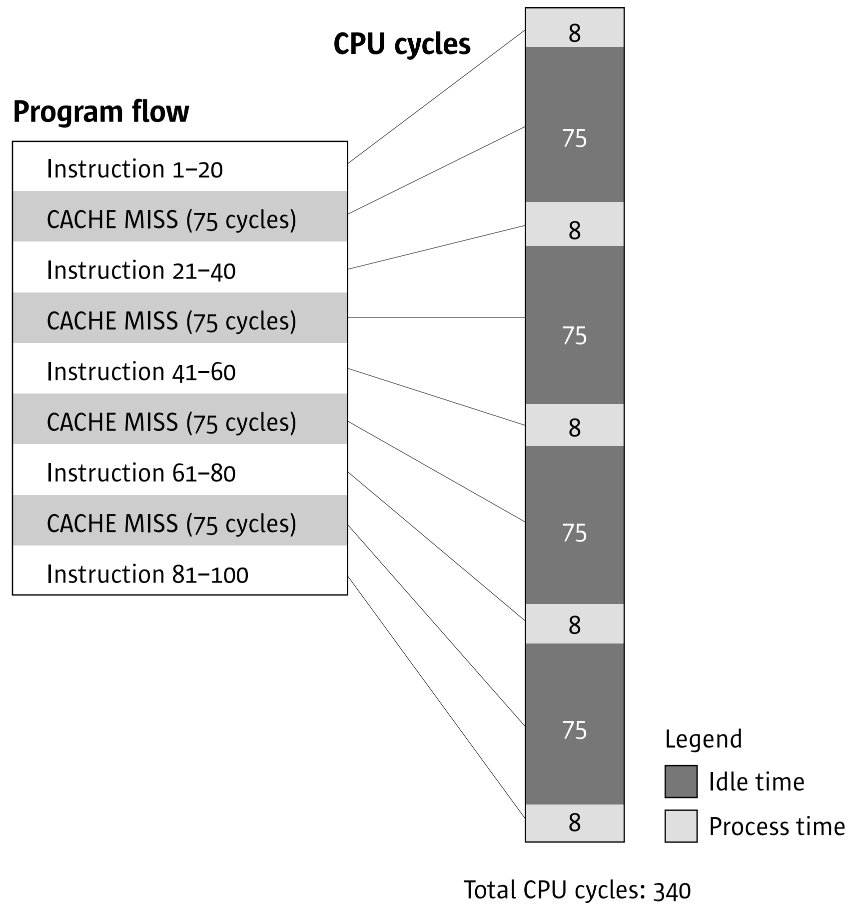
Let us calculate the total CPU cycles required for program execution for:

1. VLIW-based processor without multithreading, and
2. VLIW-based processor with multithreading (MAJC).

VLIW PROCESSOR WITHOUT VERTICAL MULTITHREADING

From our assumption before, an average of 2.5 program instructions are executed in each CPU cycle. It is to be noted here that during a cache miss, the processor is idle and waits for the contents to be loaded from the memory to the cache.

The total CPU cycles for the program when executed on traditional VLIW based architecture without vertical multithreading is shown below.



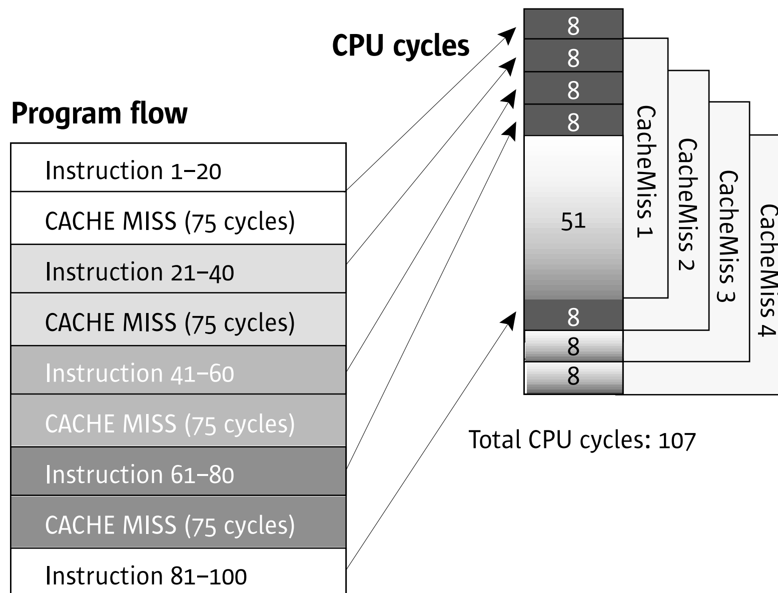
Total CPU cycles = Total process time + Total idle time = 40 + 300 = 340.

VLIW BASED PROCESSOR WITH VERTICAL MULTITHREADING

Now, let us look at how the program executes on a four-issue wide VLIW-based processor with vertical multithreading. As before, an average of 2.5 program instructions are executed in parallel in each cycle. The difference however is in the processing of a cache miss. The program can be viewed to be divided into five threads, each with twenty instructions. Upon the first cache miss, the processor saves all references to thread-1 in a set of registers from the large unified register file and starts processing thread-2. Upon the second cache miss, all references to thread-2 are stored in another set of available registers and starts executing thread-3. The process continues.

The MAJC™ architecture is designed to support a total of four threads. Hence, we see that during a cache miss, the processor is not idle but is working on a different instruction stream. This helps significantly in reducing the total CPU cycles required to execute the program. However, there may be cases when even after executing four threads, information required for processing thread-1 information is still not available in the cache. The processor in such cases will be idle, but the wait time is not substantial.

The effective CPU idle time during each cache miss and the total CPU cycles required for executing the program is shown below.



Legend

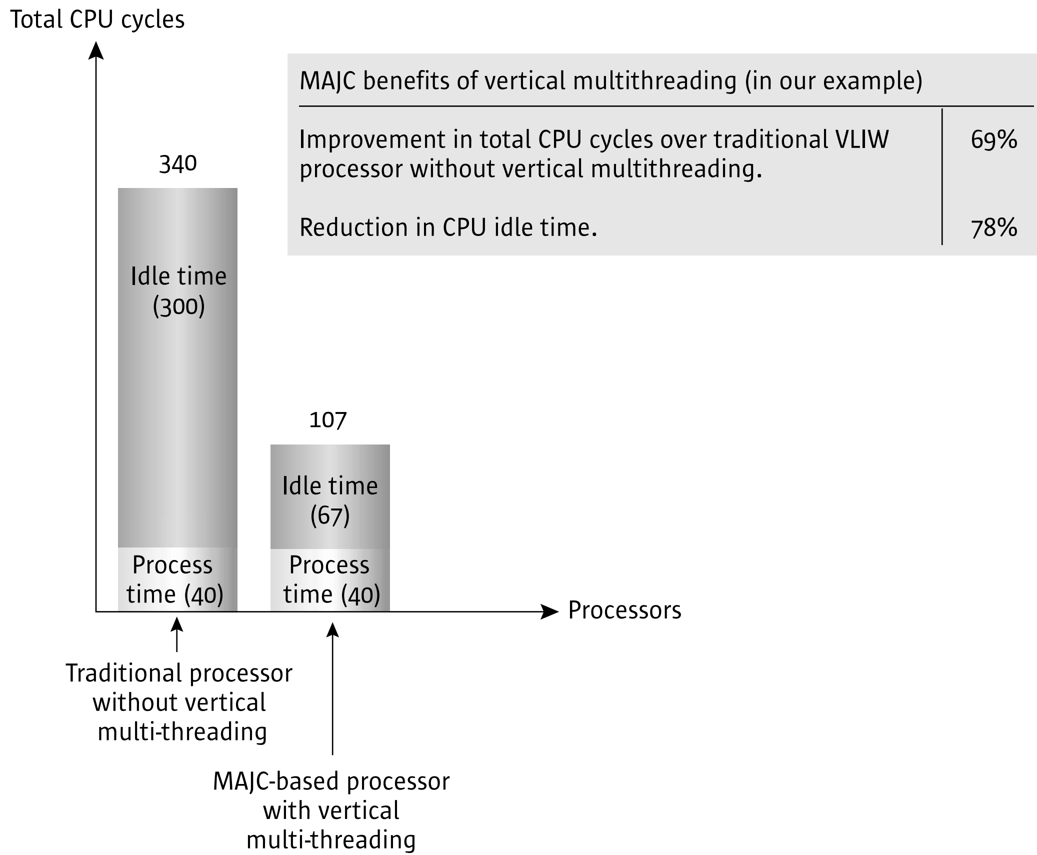
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5 (New Thread 1)
Process time
Idle time

Idle 1 = 51 cycles (Thread 2 + Thread 3 + Thread 4)
Idle 2 = 0 cycles (Thread 3 + Thread 4 + Idle 1 + Thread 5)
Idle 3 = 8 cycles (Thread 4 + Idle 1 + Thread 5)
Idle 4 = 8 cycles (Idle 1 + Thread 5 + Idle 3)

Total CPU cycles = Process time + Idle time = 40 + 67 = 107

BENEFITS OF VERTICAL MULTITHREADING IN OUR EXAMPLE

A comparison of the total CPU cycles required for executing our program is shown below.



Thus we clearly see that processors based on the MAJC™ architecture with Vertical Multithreading substantially increase CPU utilization in the presence of delays such as cache misses. This decreases overall CPU idle time and increases throughput.

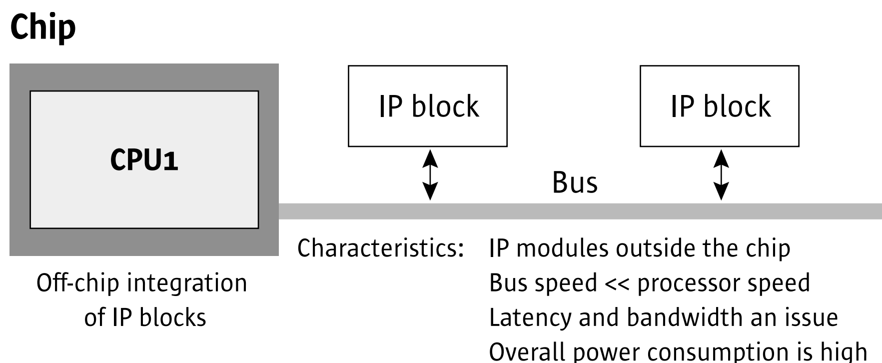
SYSTEM-ON-CHIP/MULTIPROCESSOR SYSTEM-ON-CHIP

The convergence of voice, video, and data as well as price/performance issues are driving the semiconductor industry toward integration of IP (Intellectual Property) blocks on a single chip. System-on-Chip (SOC) implementations based on processors using the MAJC™ architecture will provide substantial improvement in price/performance ratios while significantly lowering the overall system cost.

Please note that SOC is an implementation benefit using and not an architectural feature of the MAJC architecture.

Architecture Features Aiding SOC Implementation	Benefits
• Fast switch operating at processor speed connects IP blocks and CPU, providing high bandwidth and reducing latencies • Architecture supports instruction-level and thread-level parallelism. Multiple processor units can reside on the same die, sharing a common data cache • IP blocks reside on the same die and share information in real time with other IP blocks/CPU	• Real time information movement between IP blocks and with CPU • Substantial improvement in price/performance measures • Consumes less power and significantly reduces overall system cost

Increased Price and Decreased Performance Is a Problem In Off-chip Integration



In off-chip integration, IP modules (such as, memory controller/DMA controller/pre-processor etc.) share a bus outside the chip. Two basic problems arise in such a setup—price and performance.

Implementing critical IP blocks outside the chip has a direct bearing on the pin count required on a chip. Increasing the number of I/O pins required on a chip directly results in increased package costs, and subsequently, increased cost of the overall solution.

The other, more important issue in off-chip implementation is poor performance in latency and bandwidth.

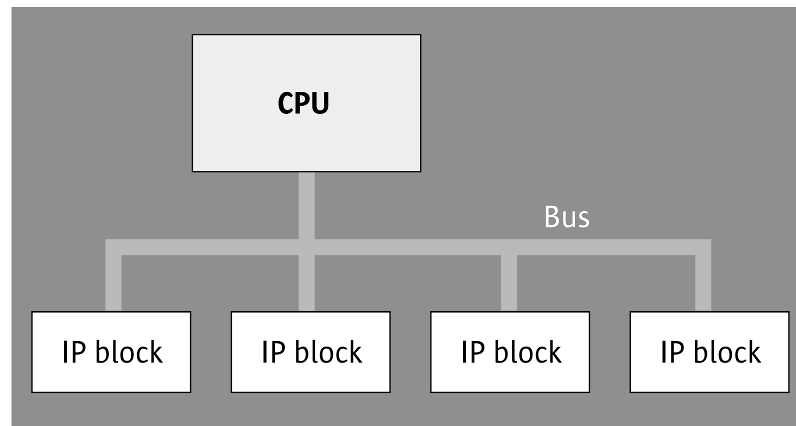
In off-chip implementations, IP modules run on a bus outside the chip. Though processors today can run at speeds as high as 500–600 MHz (as an analogy, imagine this to be the freeway speed), the buses run at speeds of 66–150 MHz (equivalent to lower speed limit in the citylimits). This latency is a bottleneck. Such slow buses directly affect the throughput of the system.

The bandwidth available on the bus is also shared by devices connected to the bus (sharing lanes within city limits). This means that as more IP modules use the bus, the available bandwidth is reduced. To minimize the bandwidth problem, wider buses can be designed. However, this will increase the overall cost of the solution.

Another disadvantage in off-chip implementation is power consumption. The more pins that are required on a chip, the more power is required to drive the signal on the wire. Hence the overall power consumption increases.

On-chip Bus Implementations Reduce Price. Performance Is Still an Issue

Chip



IP blocks on a chip, connected by a bus.

Characteristics:
- IP modules on chip
- Latency and bandwidth are still an issue
- Bus speed << processor speed
- Overall power consumption is low

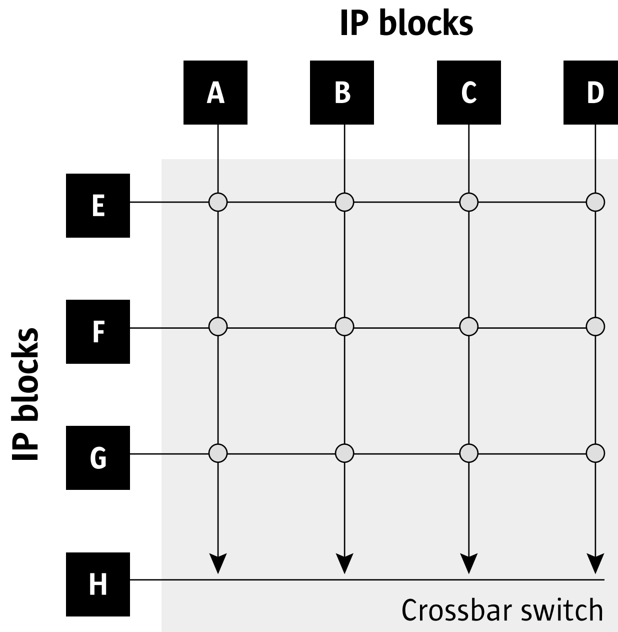
In the on-chip bus implementation, the IP blocks are present and integrated on the chip and share a common system bus. Sharing a common bus by the IP modules results in loading the bus. In case of bus contention by the IP modules, arbitration will also be difficult (no point-to-point access). Further, system buses operating at frequencies (66–200 MHz) lesser than fifty percent of current processor speeds (500–600 MHz) add to the problems of latency and bandwidth.

Today's applications involve high speed data movement. In applications that involve high end graphics or in networking, where computing at wire-speed is critical, such latencies and bandwidth limitations are not desirable. The MAJC™ architecture is designed to handle these issues.

It is to be noted that on-chip bus implementations of IP modules are an improvement over off-chip bus integrations. The number of I/O Pins on the chip are reduced and hence decreases the power required to drive the Pins.

MAJC™ Architecture: Fast Switch Operating at Processor Speed Minimizes Latencies and Provides Superior Performance

A simplified cross-bar switch (for tutorial purposes) is shown below:



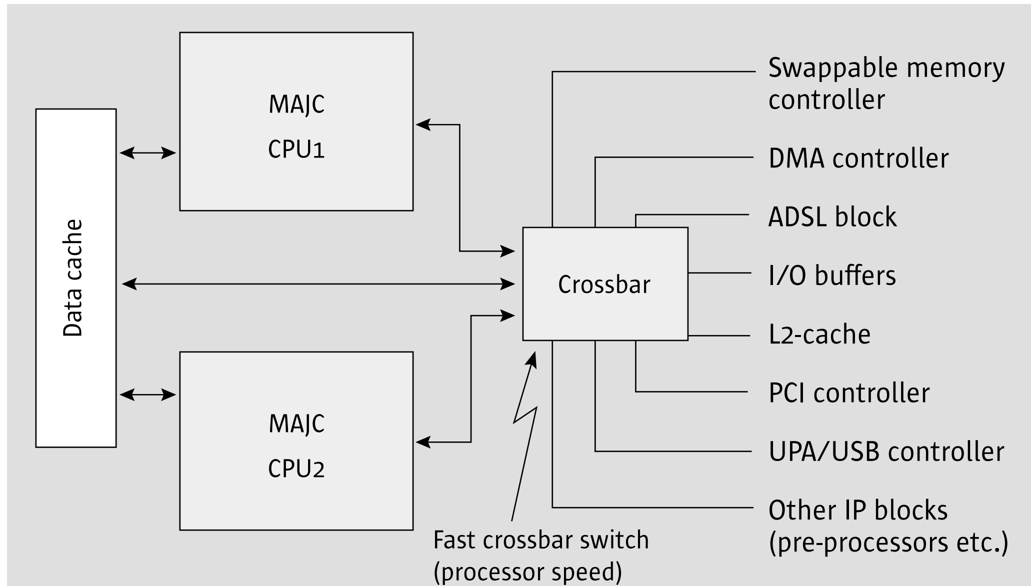
A, B, C, D and E, F, G, H are I/O blocks. A cross-bar switch in the above figure shows how individual blocks A, B, C, and D can talk to any of E, F, G, and H (actually, using a cross-bar switch any IP block can talk to any other IP block i.e. A, B, C, and D can also talk to one another). When device A talks to E, device B, C, and D are free to talk to any of F, G, and H. When the cross-bar switch operates at the processor speed, there is minimum latency in the information shared between the IP blocks and with the processor. Also, bandwidth available to communicate among the IP modules is high, as IP block A talking to E in no way impacts B talking to either F, G, or H.

MAJC architecture implementations allow IP modules to be connected by a fast switch, operating at processor speed. The choice of the switch depends on the bandwidth requirement of the IP block. If the bandwidth requirement is in the order of many Gigabytes, a fully cross-bar switch may be used. Else, a simple switch may suffice. The switch acts like a “traffic cop” and arbiters access to various IP modules. For example, if many IP modules contend for the memory, the switch can arbiter on the priority of access to the IP modules.

In the MAJC architecture, the switch runs at processor speed and operates at a much higher frequency as compared to bus implementations operating at speeds of 66–200 MHz. Hence latencies are brought down to a minimum and throughput is significantly increased.

The switches in the MAJC architecture based implementations can exchange data at a very high rate. Such data movements help in networking, high end graphic applications and in general, where computations at wire-speed is essential.

An example of a cross-bar, System-on-Chip implementation using the MAJC architecture is shown below.



On-chip crossbar integration of IP modules.

Characteristics: IP modules on chip
Crossbar switch speed = processor speed
Substantial improvement in latency and bandwidth
Low power consumption

Implementations using the MAJC architecture in a SOC configuration can obtain significant price/performance improvements due to the following factors:

- The MAJC architecture allows tightly coupled CPUs to share a common data cache (or run on a very low latency on-chip coherency bus operating at processor speed) providing large coherence bandwidth.
- The MAJC architecture allows multiple levels of parallelism in each processor, reducing the overall CPU cycles required to execute applications and increasing computational performance.
- A fast switch operating at processor speed substantially reduces latencies and enables wire speed computations.
- System-on-Chip configuration requires fewer I/O pins, lowering packaging costs and reducing power consumption.

HEADQUARTERS **SUN MICROSYSTEMS, INC.,** 901 SAN ANTONIO ROAD, PALO ALTO, CA 94303-4900 USA
PHONE: 800 681-8845 FAX: 650 969-9131 INTERNET: www.sun.com/microelectronics

SALES OFFICES

FRANCE: +33 1 30 67 53 95
GERMANY: +49-89-46008-474
ITALY: +39 039 60 55 266
JAPAN: +81-3-5717-5060
KOREA: +82-3-469-0298
PEOPLE'S REPUBLIC OF CHINA: +86-10-6803-5588 x28882
RUSSIA: +7-095-935-8411
SINGAPORE: +65-438-1888 x58677
SWEDEN: +46 8 623 9250
TAIWAN: +886-2-2514-1181
UNITED KINGDOM: +44-1276-416944

SUN™ © 1999 Sun Microsystems Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo, MAJC, Java, and We're the dot in .com are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. This Publication is provided "AS IS" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. This publication could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein; These changes will be incorporated in new editions of the publication. Sun Microsystems, Inc. may make improvements and/or changes in the product(s) and/or the program(s) described in the publication at any time.

LFC2.3 9/99