Amdahl Multiple-Domain Architecture

Robert W. Doran
University of Auckland*

Products from IBM have, for more than 20 years, taken the major share of the general-purpose mainframe computer market. This continual dominance by one supplier has meant that the interfaces between its system components have become de facto standards. Any manufacturer wishing to supply hardware or software products to the major portion of this market must recognize these standard interfaces.

The term "plug compatible" has come to apply to products designed in accordance with such de facto standard interfaces. In this article, I use the term "standard" to refer to the de facto standard interfaces followed by the plug compatibles. (Hellerman discussed the origin of these standards.[1])

Amdahl Corporation produces standard hardware and software products. Since its founding in 1970, Amdahl's main product has been a high-end processor system. Given that this system runs with standard software, it follows that it must support the current standard architecture. Indeed, the first Amdahl computers were hard-wired implementations of the IBM System/370 architecture. However, in later products, the Amdahl architecture has included many extensions to the standard architecture of the time. Here I will describe these extensions, why they were made, and how they developed.

Amdahl's architecture for its mainframe computers helps maintain compatibility with a changing standard. Its multiple-domain facility allows multiple operating systems to share one mainframe system.

Goals of architecture extensions

A plug compatible manufacturer, or PCM, can justify extensions to its architecture on two grounds. The first, and most important, is to make it easier for the PCM's products to remain compatible with the standard despite the continual extensions and changes made to it. The second is to provide better facilities to customers as a value-added alternative to IBM and to the PCM's competitors.**

Compatibility. The concept of compatibility might look straightforward, but it turns out to be surprisingly complex. The architecture with which the PCMs must be compatible, although a de facto standard, is continually being extended, and the PCMs have no prior knowledge of changes to come. When a change is made to the standard, the PCMs must also implement a change. However, any change always takes time before it is widely used—the delay in acceptance depends on the magnitude of the change and the extent of the difficulties its adoption causes to customers' operations. This natural delay means that the PCMs do not have to follow each change instantly, but can spread their response over time.

Compatibility for a PCM thus means not only compatibility with the current standard, but also the ability to follow

*Doran prepared this article while on leave, working at Amdahl Corp. in Sunnyvale, California.

**The standard mainframe business is awash in acronyms such as PCM; these are too prevalent to avoid, but I define them on first use and include them in the glossary if used frequently. Note that I do not use the term PCM here to include IBM itself, although it, too, is significantly constrained by the standards that it has created.

0018-9162/88/1000-0020$01.00 © 1988 IEEE

changes to the standard in a timely manner. The latter mainly requires careful management of the design and manufacturing processes for fast turnaround of changes, and flexibility in the design architecture to accommodate extensions. However, it also has implications for the architecture that lead to the surprising conclusion that, to be compatible in the future, the architecture should differ from today's standard.

One reason for this apparent paradox is the desirability of anticipating change. Although we cannot know the details of future extensions, the really important changes will be made by the dominant supplier to solve problems experienced by its "leading edge" customers. We PCMs can observe those problems, judge the kinds of changes needed, and set up our computers to have similar features.

Such anticipation, although not necessary to ensure future compatibility, can greatly reduce the difficulty and cost of later modifications. Our solutions to the eventual changes will surely differ from these interim measures, but, meanwhile, we will have gained experience with the design and application problems involved. Our product will have appropriate data paths and so be easier to modify. Our product might even adequately adhere to the new or revised standard without major changes to its hardware.

Even without anticipation of changes to the standard, we would want to have extensions. Compatibility requires presenting the same image to software. Not having to modify standard software to have it run on your hardware grants a highly desirable "seal of compatibility" to a PCM's products. This goal applies to the operating systems that control the computer (system control programs, or SCPs, in standard terminology). However, some sections of a computer family design are model-dependent and so need specialized software for control, such as for the details of machine checks or reconfiguration. It follows that we want to have some way of running both standard SCPs and PCM model-specific software on the same computer, with such PCM software being invisible to the other software.

Once we PCMs accept the desirability of such software, we can further develop the idea. Rather than regarding our software as equal to other software, we would naturally consider it the ultimate controller of the system because it deals directly with PCM hardware.

We must define mechanisms that allow the PCM software to coexist with the standard SCP, yet maintain the ability to assert control when necessary. The events that trigger control by the PCM software, such as on machine checks or when initially starting the computer, must be selected and the details of control switching spelled out. Obviously, we must extend the architecture of the PCM computer considerably to encompass this.

For the PCM, anything that can aid in attaining compatibility quickly is valuable. As mentioned above, design flexibility is essential; for example, having spare microcontrol storage. However, flexibility often conflicts with other goals, such as high processor speed and low cost. Given that we need PCM software anyway, there arises the possibility of our using it in attaining compatibility with the unknown future. It would effectively act as an extension to microcontrol store. This requires that we give more thought to circumstances in which control has to be transferred to the PCM code, as well as to the efficiency of the mechanisms involved. These considerations will further increase the richness of the PCM architecture extensions, which are justifiable on the grounds of compatibility alone.

Added value. A PCM supplier must select with care extensions made for added value so as to satisfy a number of seemingly conflicting criteria. Because customers live in the standard world, it is best if any extensions appear to supplement or extend what the dominant supplier offers. Innovations that obviously differ and that appear to make customers dependent on the alternative supplier, although possible, are difficult to introduce because the justifiably conservative majority of customers will resist them.

Innovations should also be unobtrusive. For example, it is important that the PCM-specific controlling software look built-in to the hardware of the machine, that it not complicate operation, and that it definitely not appear to the customer as another SCP to be mastered.

It is thus advisable that extensions align with what the dominant supplier does.

Glossary

Note that terms used to describe the Amdahl architecture are italicized.

channel: processor for performing I/O in 370 architecture

CMS: *Conversational Monitor System*—IBM time-sharing system, part of VM

control state: mode of operation for macrocode

CP: control program—kernel of VM

CPU: central processing unit—processor

CR: control register

domain: set of resources controlled by an SCP

fast assist: mode of operation for instruction emulation

GPR: general-purpose register

guest: a virtual computer or a program running under a host

host: SCP that provides a virtual-machine environment

macrocode: hardware-specific code *used to implement an architecture*

MDF: Multiple Domain Facility—architecture to support multiple SCPs in production operation

MVS: Multiple Virtual Spaces—the main IBM SCP

native: operation of a program not as a guest

PCM: plug-compatible manufacturer—manufacturer of products to de facto standard interfaces

PSW: program status word

SCP: system control program—an operating system

SIGP: signal processor—inter-CPU signaling instruction

system register: control register for control state

System/370: the architecture implemented in IBM mainframe computers

user state: mode of operation for a guest SCP

VM: virtual machine—an IBM SCP

XA: Extended Architecture—*extension to System/370*

One possibility that looks safe is to solve problems (such as addressing limitations) that clearly must be dealt with and to make the solutions available to the PCM customers. Unfortunately, this approach encounters a major difficulty: when the dominant supplier solves the same problems, the PCM extensions will likely become redundant because PCM customers will tend to prefer the new standard approaches, other considerations being equal.

From the above arguments we can conclude that the PCM must be different yet appear to be the same. Although actually a reasonable goal for design features (such as making the PCM product faster), it is indeed quite a restriction for architecture. If that were not enough, even more constraints arise just from business considerations. In particular, the PCM wants to obtain extensions—especially those not required for compatibility—at low cost. Rather than using the hardware involved to provide an advantage that is merely projected, the PCM could use it for more obvious benefits, such as making the machine faster, or leave it out altogether, thus making the machine less costly.

Given all the restrictions under which PCMs operate, we might question whether any added-value extensions are reasonable. However, consider the main direction—taken by Amdahl—of allowing multiple SCPs to run on a single computer. IBM itself supported such a facility, but Amdahl could take an improved, fundamentally different approach unlikely to be followed by IBM in the short term. Customers could always replace the Amdahl feature, albeit at the cost of purchasing multiple computers or by reverting to the more cumbersome IBM solution, so the approach did not commit customers to the PCM permanently. It would not require changes to SCPs and so had no impact on compatibility. Moreover, it would appear as a built-in hardware feature, easy to operate. It thus neatly satisfied all of our external criteria and constraints.

Because, for compatibility reasons, Amdahl needed to provide support for controlling software outside the realm of the standard SCP, we could argue that the hardware cost to support multiple SCPs added little to the total cost. So, we could convince ourselves that support of multiple SCPs also satisfied our internal constraints. From this conviction, Amdahl developed an architecture based on the concept of simultaneously executing multiple SCPs, with extensive attention to details that assist in maintaining compatibility.

Because macrocode is under our control, the architecture that it sees does not have to be the same as the standard.

Support of multiple operating systems

Rationale. Given that one SCP involves quite enough complexity all by itself, it might surprise those unfamiliar with the large mainframe world that there is interest in running more than one operating system on a single computer. In fact, a mainframe computer center that supports a large number of users gains many advantages. Some of the benefits the customer might obtain include:

- Customers can test and install new versions of SCPs during normal working hours without disrupting production operation.
- Customers can run different SCPs, specialized for different applications, on a single computer.
- Customers can run multiple copies of the same SCP, each tuned for a different class of application.
- Rather than modifying old programs to allow them to run under new versions of SCPs, customers can operate both the old and new versions of the SCPs. If necessary, they can make conversions gradually.
- A separate SCP might provide sufficient security in some applications that the customer can avoid purchasing a special machine or restricting certain periods of operation.
- Given that software failures occur more frequently than do hardware failures, a backup SCP can provide improved availability.
- Disparate workloads, resulting from mergers, acquisitions, or rationalization, can be consolidated on the same system.
- Customers can obtain considerable savings by only having to license one copy of an SCP and the software that runs under it.

Some benefits from multiple SCP operation accrue more directly to the vendor of large systems:

- One large system can replace multiple small systems, thus creating new sales opportunities.
- It is easier to introduce an entirely new SCP, because the customer's operation is not immediately dependent on the new SCP.
- Vendors can use special SCPs for diagnosis and measurement, running them in parallel to the SCPs being monitored.

The VM SCP. Of all the above advantages, the most exigent from the customer's point of view would be that of testing new releases of SCPs. This need led IBM itself to support multiple SCP operation. This interesting story has been described at length elsewhere,[2-4] so I will only recapitulate the main points here.

One of the three main SCPs supported by IBM for large mainframes is VM/370. This was developed, and flourished, because it provided the Conversation Monitor System, or CMS, time-sharing subsystem.[2] VM itself was designed with the idée fixe that time-sharing is best implemented by providing each user with the illusion of complete control of the computer system. Under this approach, the user is provided with a virtual machine environment indistinguishable from the original computer system.

Naturally, SCPs can run (as guests) in a virtual machine under VM (the host), so the ability to run multiple SCPs is immediately available. A guest operates using virtual resources as opposed to the real resources controlled by the host—there may be many more virtual resources, such as processors and main storage, than real resources.

This approach has the disadvantage that the performance of an SCP running as a guest is severely degraded when compared to performance running on the real machine (native operation). For VM to maintain control of the system, the guest SCP has to operate in nonprivileged mode with all supervisor-state instructions trapped by VM, checked, and simulated. Additionally, because the guest SCP believes that it controls the whole computer, any resource mappings that it sets up (such as page tables) have to be intercepted and modified by VM. (Goldberg discussed the general topic of virtualization of resources.[5])

As mentioned above, the ability to run test SCPs was so important to some users that they did use VM to run other SCPs, accepting the loss of performance involved. IBM noticed this class of troubled users and responded by enhancing the machines running VM to make them perform better, using various architecture extensions called assists.[3] Eventually, with the features called Virtual Machine Assist plus Preferred Machine Assist (on the System/370) and Interpretive Execution Facility plus SIE Assist[4] (on the System/370-XA, the successor architecture to the System/370), it became possible to run one selected SCP (the "preferred" guest) under VM with performance very close to native operation. This made it possible to use guest SCPs for production operation.

These enhancements were most advantageous to IBM when migrating customers from the 24-bit-addressing MVS to the 31-bit-addressing MVS/XA. IBM provided customers with the VM/XA Migration Aid, which comprised a "cut-down" version of VM that could run both old and new systems simultaneously.

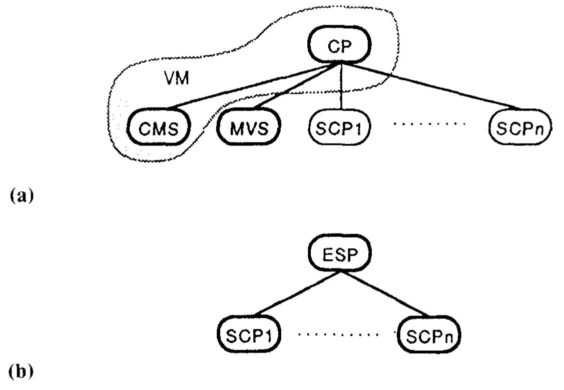

Figure 1. Contrasted models of operation, showing (a) IBM with MVS as preferred guest under VM and (b) Amdahl multiple domain operation. Bold outlined systems can be used for production operation.

The multiple domain facility. The Amdahl architecture was specified in 1977, long before the above events, which took place in the early 1980s. We at Amdahl had also noticed the customers' needs, but we didn't find the VM approach attractive because it would involve producing a full SCP and certainly would run counter to the criterion of being unobtrusive. Amdahl software engineers explored a different possible approach, which eventually led to a product called VM/PE (for Performance Enhancement). This involved running the MVS operating system as if it were in control of the whole computer by placing it in the low portion of the real address space and not telling it of the existence of the high address space where VM was placed. It required the addition of software to MVS so that control could be given to VM on the occurrence of interruptions.

The security of the approach depended on the reliability of MVS code. Although this was fine in practice, we did not want a situation where each SCP could interfere with another, nor one that needed modifications to a standard SCP. Amdahl's VM/PE is acknowledged and described in Bean and Gum's patent[6] for IBM's Preferred Machine Assist, which continued the same approach under VM, but fixed some of its problems.

It was also natural for Amdahl to extend the VM/PE approach but, rather than concentrating on improving MVS under VM, it made more sense for us to regard the two as equals and to put the code that controls the allocation of the hardware outside of both. This led to our approach, contrasted to the IBM approach in Figure 1. Our approach added another level of control to the system, containing a controller that came to be termed macrocode (Amdahl uses macrocode to refer to both the class of code and the control entity itself). Macrocode, supported by necessary architecture changes, has the following goals:

- Be ultimately in control of the computer hardware.
- Share the hardware among multiple SCPs with minimal overhead.
- Appear to users to be built-in, part of the hardware.
- Assist with providing a compatible image to standard SCPs.
- Provide secure boundaries between independent SCPs.

Note that, although macrocode has some of the functions of an SCP like VM, its rationale and goals are quite different. Consequently, quite different architectural solutions are called for. Because macrocode is under our control, the architecture that it sees does not have to be the same as the standard—which is an astounding degree of freedom to give a PCM. Customer programs never see the architecture, so there is no external compatibility reason for it to be the same from model to model, or for it not to evolve over time. The architecture can be very specific to the hardware and defined at a level of detail that would be entirely inappropriate for general programming. This intermediate status accounts for the coining of the marketing term macrocode to describe coding at this level.

Outline of architecture

This description applies generally to the architecture developed for the Amdahl 580 and 5890 series computers, despite differences in the details of different models. Initially, only the System/370 architecture was supported, but later the Amdahl architecture was extended to allow simultaneous execution of SCPs using System/370 or System/370-XA. Most of the Amdahl architectural extensions apply to processors, called CPUs in the mainframe world.

The description below should be understandable to the general reader, but complete appreciation requires some knowledge of the IBM System/370 architecture.

Amdahl tries to support the model of operation displayed in Figure 1b, where there is one controller called macrocode and multiple SCPs that operate independently, with the goal of minimal overhead so that several of the SCPs can be used for production operation. We take this to imply that, if there are no more virtual CPUs than real CPUs, and macrocode is not invoked for compatibility reasons, then there should be essentially no diminution of performance compared to native operation of the SCP.

Resource sharing and mapping. The total system is shared among the multiple SCPs and macrocode. Let's first see how the system components operate on behalf of macrocode and each SCP, how the SCPs are protected from each other, and how the SCPs are provided with the illusion of total control.

We need to distinguish between the operation of the hardware for SCPs and for macrocode. We introduce a new state of operation for CPUs, called control state, for macrocode, which we contrast with operation for SCPs, called user state. In both states, the system operates as a variant of the general System/370 architecture. The term *domain* refers to the set of resources used by an SCP in the user state, and the term *system* refers to the entire hardware resource as seen by macrocode.

Each CPU includes a new set of registers, used by macrocode to control domain operation, called system registers (see Table 1).

Table 1. Control fields held in system registers.

Control Fields
System addressing limit
Domain main-storage base
Domain main-storage limit
Domain channel-map pointer
Domain number
Domain CPU address
Domain prefix
System timer
TOD clock offset
State-switching interruption mask
System interruption mask
CR load mask
PSW status-change mask
Feature control word
Fast-assist base
Domain CPU status
Domain compare address

Main storage divides into sections allocated to the control of one domain or to macrocode. Macrocode uses the actual system main-storage addresses and can address any part of main storage. However, it normally operates out of low storage and, for reliability reasons, is restricted in its use of system main-storage addresses by the system addressing limit. System main-storage addresses are always 31 bits in length.

For domains, the system supports both 24-bit and 31-bit addressing. The block of main storage currently being used is delimited by a base-bound mapping defined by the domain main-storage base and limit. All domain main-storage references are checked against the limit and converted by addition of the base to system addresses (see Figure 2). Although simple, this method is entirely appropriate where memory is shared among SCPs; they do not come and go like processes and are seldom reconfigured. Note that this domain main-storage mapping is in addition to, or on top of, the virtual-storage mapping of the System/370.

In the System/370 architecture, special processors called channels perform I/O. I/O channels are each dedicated to a domain or to macrocode. A domain channel map (see Figure 3) held in main storage, pointed to by a system register, maps domain channel addresses to system channel addresses in a general manner (later modified to apply to the subchannel addresses of the XA architecture). Domain numbers distinguish the domains. When an I/O operation is initiated, the state of the CPU indicates which domain (or macrocode) is active so that it is possible to ensure that only a CPU operating on behalf of that domain is interrupted on completion of the operation. The channel processor, operating independently of the CPU, receives details of the memory mapping from system registers when the I/O operation is initiated. Thus, initiation or completion of an I/O operation requires no additional software intervention.

CPUs are not dedicated to SCPs, but can be time-multiplexed between the domains and macrocode. The System/370 architecture specifies a system comprising a number of CPUs, each identified by a CPU address. A domain CPU address held in a system register allows any real CPU to act as any domain CPU. The interprocessor signalling instruction SIGP (for signal processor) uses the domain CPU address to locate the correct processor to signal. The old System/370 architecture also used the domain CPU address in I/O, where an I/O interruption had to be serviced by the CPU of origin.

The mappings defined above allow multiple domains to operate with little overhead. As described, once SCPs are set up, running them under macrocode involves no significant overhead. Translation steps involved with I/O and inter-CPU signaling occur too infrequently to measurably affect performance. In modern fast computer designs, the extra level of main-storage mapping is invoked only for the few references for which translation is not bypassed by a translation lookaside buffer. So, the main-storage mapping also has no deleterious effect.

Multiplexing of CPUs. The situation is more complicated than just described. Macrocode has to multiplex the real CPUs among all the domains when there are not enough real CPUs to dedicate one to executing each domain CPU. Furthermore, we use macrocode to aid compatibility and, in fact, to provide part of the virtual machine illusion in circumstances that occur so infrequently as to not warrant implementation in hardware.

Consequently, transitions between macrocode and SCP are much more frequent than transitions between SCPs. Above, I gave the impression that CPUs would be time-multiplexed between macrocode and SCPs. This holds true for resources not used by macrocode, such as floating-point registers, but other resources needed by both macrocode and SCP are replicated. There are both system and user general-purpose registers, or GPRs, as well as both system and user prefix registers. (The original System/360 was essentially uniprocessing and used the bottom 4 kilobytes of real storage as an extension of the CPU registers. The prefix register was introduced to the System/370 to relocate that area to permit more than one CPU.) The System/370 control registers (or CRs), which contain fields affecting system operation, occupy an intermediate status. Most are not needed in control state, since macrocode does not use virtual addressing and thus does not need the associated controls, but needed fields are duplicated in system registers.

Now, let's look at the extensions to allow macrocode to multiplex the CPUs between SCPs and itself. We expect that control does not pass from SCP to SCP, but always through macrocode. Let me outline the sequence taken when control passes from macrocode to SCP and back.

Macrocode loads the system registers that define the domain in which the SCP operates. The state of execution of the CPU is set properly, meaning that the domain GPRs are loaded to reflect the current state of execution of the domain on this CPU. Macrocode then executes a special instruction to give control to the SCP.

For time slicing, macrocode uses a special system timer to regain control. The system timer is a direct analog of the System/370 CPU timer. When it counts down to zero, it causes an interruption that gives control to macrocode. Control can also be returned to macrocode in other circumstances that indicate that the SCP has ended its time slice early.

One detail of interest when discussing CPU multiplexing is the System/370 time-of-day clock. The TOD clock is a binary counter, incremented at least once a microsecond, that wraps around in 143 years. Different SCPs might operate their clocks in different epochs, giving different meanings to time zero. Macrocode does not use the TOD clock to cause interruptions, so there is no need for two TOD clocks in each CPU. However, it is important in a multiprocessor system that TOD clocks on separate CPUs not lose synchronization. So, we would not want the TOD clock to change when a domain is dispatched because that might cause errors to accumulate in the TOD value. Consequently, the Amdahl architecture provides a TOD offset that is subtracted from the system TOD clock to give the TOD for a particular domain.

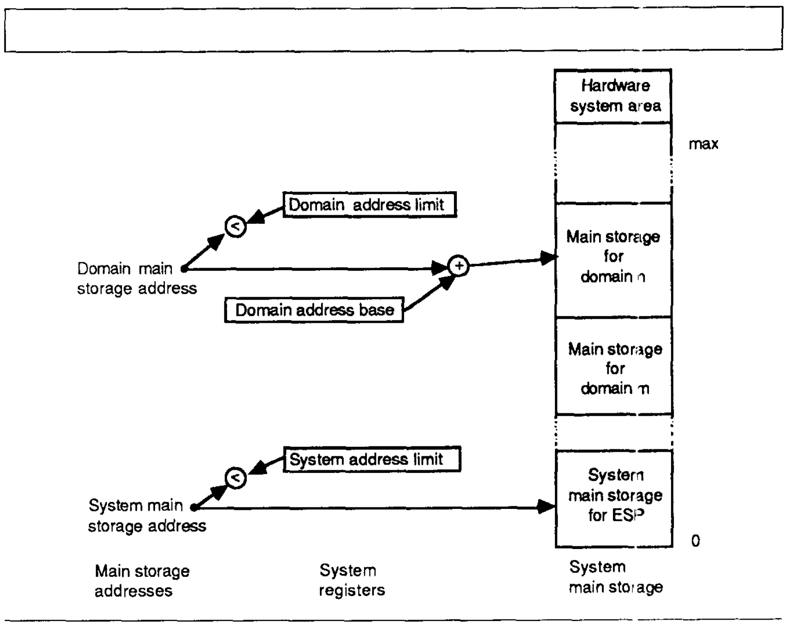

Figure 2. Main storage organization.

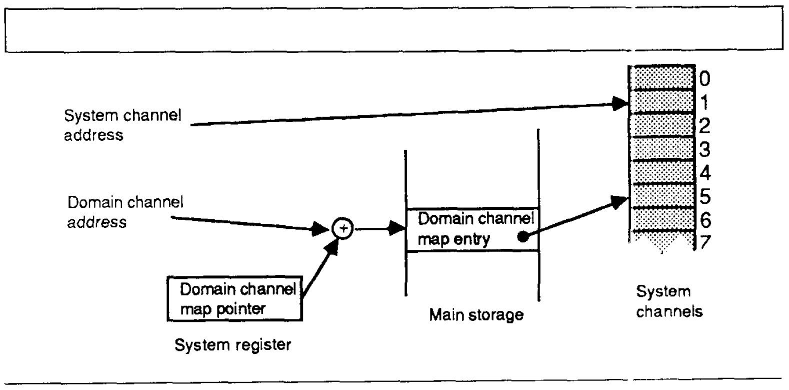

Figure 3. Domain channel maps.

Interruptions. The above outline immediately raises many questions about the interruption process. The interruption mechanisms needed for multiplexing make up a small part of what we need to prepare for compatibility. Moreover, they are imbedded in the more general architecture, which is best considered separately. Amdahl includes most of the features described immediately below to allow a fast response to changes in the standard architecture; they do not operate under normal circumstances.

In the System/370, the current instruction address is stored with other control fields and bits in a special program status word, or PSW, register. An interruption involves saving the old PSW in main storage and loading a new one. We are interested here in state-switching interruptions that transfer control from a domain SCP to macrocode. We can make an interruption state-switching by preceding it with a switch from user state to control state. That changes operation from domain to system addressing, so that the PSW swap takes place in system main storage. When the interruption entry completes, macrocode specific to the interruption class executes.

In preparing for compatibility, we must ensure that macrocode can gain control upon the occurrence of any situation on which one could define an architectural extension. The System/370 interruptions fall into this category. They can be specified as state-switching by bits in the state-switching interruption mask. Some specific interruptions, such as restart, always cause a switch in state. The state-switching interruption mask also includes bits that override or replace System/370 bits used to mask interruptions (see Table 2). Some interruptions that occur not on behalf of the SCP but for macrocode can also cause a state switch, such as an I/O interruption for I/O initiated by macrocode.

Other interruptions do not belong naturally to any of the System/370 classes because they are used only to switch state. These are assigned to a new system interruption class, controlled by the system interruption mask. This includes the system timer interruption mentioned above and some monitoring interruptions.

Some events that occur in the domain need to be monitored by macrocode. We always face the possibility of extended interpretations of the meaning of System/370 control registers, in which case macrocode needs to know when the registers are set by an SCP. By providing a CR load mask, we enable the CRs to be individually singled out for monitoring.

Alteration to mask bits in the PSW is an important change. The event of an SCP entering the wait state corresponds to the turning on of the wait bit in the PSW. Another potentially interesting event occurs when the SCP enables for some class of interruptions by changing a PSW mask bit; macrocode can only emulate an interruption in an SCP's domain when the domain CPU is enabled to accept that interruption. These changes of status are monitored by bits in the PSW status change mask. When the monitored bits change as specified by the mask, an immediate system interruption results. A system interruption code placed in storage summarizes the cause of a system interruption (see Table 3).

Table 2. Bits of state-switching interruption mask.

Bits
Machine check mask
Check stop control
Recovery report mask
Degradation report mask
External damage report mask
Warning mask
External interruption switch
I/O interruption switch
SVC interruption switch
Page translation exception switch
Program interruption switch
Interval timer mask
PER override

Table 3. Bits of system interruption code.

Bits
System timer
PSW status change
CR load
Single-step completion
CPU address compare
External address compare
Interruption from fast-assist mode

Fast-assist mode. We can detect the most important type of extension, the introduction of new instructions, by the occurrence of a program interruption for an invalid operation code. At one time, we thought that we could use state-switching interruptions to gain control to emulate new instructions. However, the interruption process turned out to be difficult to implement so that it would be fast enough for this purpose. Instead, we decided to introduce a substate of control state, which became known as the fast-assist mode, that had a predetermined, quickly invoked state of execution. We designed the fast-assist mode to make it possible to invoke emulative code for new instructions with minimal overhead. It is closely associated with the design architecture of the particular CPU.

Any operation code not defined will, when executed, cause an entry into fast-assist mode. Some defined instructions can be selected by various control fields, such as the feature control word, to also be assisted. Other instructions can be executed by hardware in frequent cases, but infrequent cases cause assist entry. For example, with the SIGP instruction, orders such as restart are never executed directly in user state.

Fast-assist entry switches to control state with a predetermined real-system addressing mode. The domain address of the instruction immediately following is placed in a system GPR, so that macrocode knows what to emulate. Additionally, the effective addresses appropriate to the instruction type—already evaluated—are placed into the system GPRs. The entry resembles a jump more than an interruption. The state of execution of the SCP, including its PSW, is retained in the CPU so that a special jump instruction can quickly return control to the SCP.

For any domain, we can specify the instruction operation codes that not only will cause entry to fast-assist mode, but will do so through a vectored branch to arrive at code specific to each instruction. Additionally, for each operation code, we can specify the application of certain tests to the instruction, such as whether the instruction is valid for the SCP's mode of execution and whether the operands are appropriately aligned. If an instruction fails a test, bits placed into a GPR define the reason for the failure. Only instructions that pass these validity tests have a vectored entry. Others share a common entry, so that we know vectored instructions to be immediately executable. The net effect of all these features is minimized overhead of entry to, and exit from, emulative code for the instructions most frequently emulated.

We make the fast-assist entry branches relative to the fast-assist base so that we can have macrocode specific to each domain and thus support, for example, both System/370 and System/370-XA.

Emulation assistance. Whether emulative code is entered through vectored fast assist, normal fast assist, or interruption, emulation will require that macrocode be able to inspect and alter domain resources. Time-multiplexed resources encounter no particular problem, but where resources are duplicated, or there are mappings, we would want further architectural assistance because the addresses used by macrocode normally apply to system resources. The extensions made are different, but appropriate, for each type of resource.

In control state, instructions use the system GPRs. User GPRs can be manipulated using new instructions to move values between the system and user GPRs and also to store and load multiple user GPRs to and from system memory. Table 4 lists these and other new instructions.

Macrocode programmers would like to use many of the instructions to refer to domain main storage. Ideally, we would like to have the ability to specify that any main-storage address used in control state is either a system or domain address, but we know of no easy way to modify the System/370 instruction format to distinguish the address type. The first solution we tried specified a useful subset of instructions and provided new variants referring to domain main storage from control state. We selected these instructions because we expected them to be frequently used (such as OR-Immediate) or because the effect would otherwise be difficult to program (such as Purge Translation Lookaside Buffer and, for synchronization, Compare and Swap). Later, we made a nice extension where we selected some of the system base registers as always generating domain addresses.

Of course, many modes of addressing are possible when referring to domain main storage. How we specify the addressing mode depends on whether the CPU is in control state or fast-assist mode. One system register contains a copy of the control fields in the user-state PSW, called the domain CPU status. The status loads automatically when macrocode is entered by interruption, but also can be changed by macrocode to alter addressing modes. In fast-assist mode, the type of addressing used is predetermined for system addresses. For domain addresses, it is still controlled, in the main, by the retained user-state PSW.

For I/O operations, the instructions executed in control state usually refer directly to the system channel or subchannel. The domain channel map is still available, so new instructions are provided to allow a channel program in domain storage to be initiated from control state. Variants of the Start I/O domain instruction allow the domain SCP to receive resulting I/O interruptions, or we can specify that all I/O interruptions cause a switch of state to macrocode control. The domain channel map has a bit for each address that will allow execution of I/O instructions for each individual channel or subchannel to be fast assisted. Thus, all levels of involvement in I/O are possible, from none at all to having macrocode do the whole job.

The most frequent SIGP instructions execute directly in the user state. The infrequent cases cause entry into fast-assist mode. Because these are infrequent, the SIGP in system state uses only system CPU addresses (except for two extended orders to sense and reset a domain CPU).

Measurement and monitoring. We included some monitoring features to emulate the functions of the console in the System/370 architecture. The console can single-step the machine and also halt when a specific main-storage address is used. We provided these features on a domain basis, but, rather than stopping, a system interruption occurs. We used the domain compare address here. Measurement instructions are used solely for performance evaluation within Amdahl.

Development of the Amdahl architecture

The first machine with the multiple domain facility built in, the Amdahl 5860 was first delivered in mid-1982. Initially, it operated compatibly with the old System/370 architecture without using any extensions.

After the 5860 shipment, Amdahl received its first major compatibility challenge when IBM released details of the System/370-XA architecture. This had been threatened for some time and we expected that we would have to make hardware alterations to data paths or implement frequently executed instructions.

Table 4. Added macrocode instructions.

Domain Main Storage Variants
AND Immediate Domain
Compare Double and Swap Domain
Compare and Swap Domain
Invalidate PTLB Entry Domain
Load Real Address Domain
Load System from Domain
Move Characters System to Domain
Move Characters Domain to System
OR Immediate Domain
Store System to Domain
Test Under Mask Domain
Instructions to Implement XA
Request Subchannel Process
Request Channel Monitor
Request Set Address Limit
Request Interruption Deletion
Test I/O Subchannel Status
Control Instructions
Enter Fast Assist Mode
Load PSW and Return
Load System Registers
Load and Test Registers System from User
Load and Test Registers User from System
Purge Domain Maps
Purge System Maps
Resume User State
Set Domain Controls
Set Channel
Set Channel Domain
Start I/O Domain Mandatory
Start I/O Domain Optional
Store Domain Controls
Store System Registers
Measurement Instructions
Test I/O Channel Status
Write Hardware Measurement Command
Write Hardware Measurement Data

The XA architecture did force us to make some hardware changes and implement some instructions (such as the 31-bit address branch instructions), but we emulated all of the new XA I/O instructions on the 580 using macrocode. We encountered one unexpected flexibility—we could emulate some new instructions partly in hardware but mainly in macrocode. Some of the Amdahl instructions circa 1984 (see Table 4), such as Request Set Address Limit, were to allow implementation of the extended architecture. We paid a slight performance penalty for partial emulation of XA, but the speed of reintroduction of compatibility was as we had hoped.

Because we used macrocode to attain compatibility, it became a necessity. All machines Amdahl has shipped since have required macrocode to operate. The next computer designed, the 5890, implemented more XA architecture in hardware, particularly in the I/O area, reducing the performance hit to negligible levels. Since all 5890s would run under macrocode, it became attractive to use macrocode as a permanent method of implementing infrequently executed instructions. Even with only one SCP executing, macrocode is always involved in the interpretation of some instructions (such as most tracing instructions).

Further extensions to the IBM architecture have—to the extent that they are important to the customer base—been absorbed and emulated. Of particular interest was IBM's introduction of expanded storage—a large address space separate from main storage and addressed as 4K pages only. Given that Amdahl systems had a large main storage, we simply used a portion of system main storage as expanded storage, with macrocode doing ordinary memory-to-memory moves to emulate paging to and from the expanded storage.

Perhaps the most gratifying result of the Amdahl architecture has been the acceptance of multiple-domain operation. On the 580 and 5890 machines, this is offered as the added-value Multiple Domain Facility, with which users can run up to four domains. Although we knew that many customers would find MDF useful, we have been surprised by the extent of its use. As of this writing, the MDF feature is used by 50 percent of Amdahl's customers and is included on 75 percent of all new shipments. It is perhaps fortunate that Amdahl introduced MDF at a time when mainframe performance was growing at an unprecedented rate (due to multiprocessing "within the cabinet"), thereby enabling many customers to consolidate operations on the one computer. MDF offers a much simpler consolidation than requiring systems to be merged under a single SCP.

Another interesting development has been the convergence of ideas in this area. The major Japanese manufacturers, Fujitsu[7] and Hitachi,[8] have developed their own architectures to allow multiple production SCPs. IBM has improved its VM assists to allow multiple production guest SCPs. All these developments employ a user-visible, VM-like SCP. However, the most recent development, IBM's introduction of the PR/SM (Processor Resource/Systems Manager[9]) feature, supports multiple domains using a hidden controller (the architecture is based on the assists developed for VM/XA). It now appears that multiple domain operation has become a permanent feature of large mainframes. □

Acknowledgments

So many people have been involved with the Amdahl architecture that the following list cannot begin to be complete. My apologies to those not mentioned. However, particular credit is due to the other architects, Art Willis, Bob Lent, Ken Weng, Nai-Ting Hsu, Kai Wong, and Arno Krakauer; to the managers at various times and levels whose support caused the project to proceed, Dave Anderson, Kornel Spiro, Harold Shattuck, Howard Honig, Richard Bishop, Lloyd Dickman, David Morgenthaler, Gene White, and Gene Amdahl; to the hardware designers who implemented it all, in particular Bob Maier, Sherman Lee, Uli Spannagel, Gary Woffinden, and Ron Kreuzenstein; and, finally, to the macrocode programmers who made it work, Steve Harrold, John Andoh, Jim Smullen, Kevin Ewert, Joe Tumminaro, and Bob Slate. I also would like to thank my colleagues at Amdahl for their assistance in reading this article and for their many valuable suggestions. Finally, I would like to thank Amdahl Corp. for support during the writing of this article and the European Organization for Nuclear Research (CERN) for support during its preparation.

References

1. H. Hellerman, "The SPREAD Discussion Continued," *Annals of the History of Computing,* Vol. 6, No. 2, April 1984, pp. 144-149.
2. R.J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development,* Vol. 25, No. 5, Sept. 1981, pp. 483-490.
3. R.A. MacKinnon, "The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines," *IBM Systems J.,* Vol. 18, No. 4, 1979, pp. 18-46.
4. P.H. Gum, "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM J. Research and Development,* Vol. 27, No. 6, Nov. 1983, pp. 530-544.
5. R.P. Goldberg, "Survey of Virtual Machine Research," *Computer,* Vol. 7, No. 6, June 1974, pp. 34-45.
6. G.H. Bean and P.H. Gum, "Method and Means for Switching System Control of CPUs," US Patent No. 4,494,189, Jan. 15, 1985.
7. S. Kanada et al., "Virtual Machine System," US Patent No. 4,400,769, Aug. 23, 1983.
8. H. Umeno and S.Tanaka,"New Methods for Realizing Plural Near-Native Performance Virtual Machines," *IEEE Trans. Computers,* Vol. C-36, No. 9, Sept. 1987, pp. 1,076-1,087.
9. IBM, "IBM 3090 Processor Resource/Systems Manager (PR/SM) Feature," IBM product announcement 188-039, Feb. 15, 1988.

Robert W. Doran is an associate professor of computer science at the University of Auckland in New Zealand, where he serves as head of the department. From 1976 until 1982, he worked at Amdahl, where he became a principal computer architect, responsible for the specification of the architecture of the 580 series computers.

Doran's research interests lie in the areas of computer architecture, logic circuits, and computer history. He was educated at Canterbury University in New Zealand and at Stanford University in California.

Readers may contact the author at the Dept. of Computer Science, University of Auckland, Private Bag, Auckland, New Zealand.