Within the 68020

Besides new instructions, the 68020 has additional hardware features – cache memory, dynamic bus sizing and pipeling.

DAVID BURNS AND DAVID JONES

Adding a coprocessor enhances the main general purpose processor by incorporating new instructions, registers and data types into the system without overloading the main processor.

Interfacing between the main processor and coprocessor is not noticed by the user i.e. the programmer need not be aware that a separate piece of hardware is executing some of the program-code sequence. In the 68020 microcode within the device takes care of coprocessor interfacing so that any coprocessors appear as a natural extension to the main processor architecture.

Using devices without a coprocessor interface such as the 68008, 68000, 68010 and 68012, communication between the main c.p.u. and coprocessor is possible by observing the correct sequence of coprocessor primitives necessary for the interface. These primitives are a method of passing commands and data between the main processor and coprocessor.

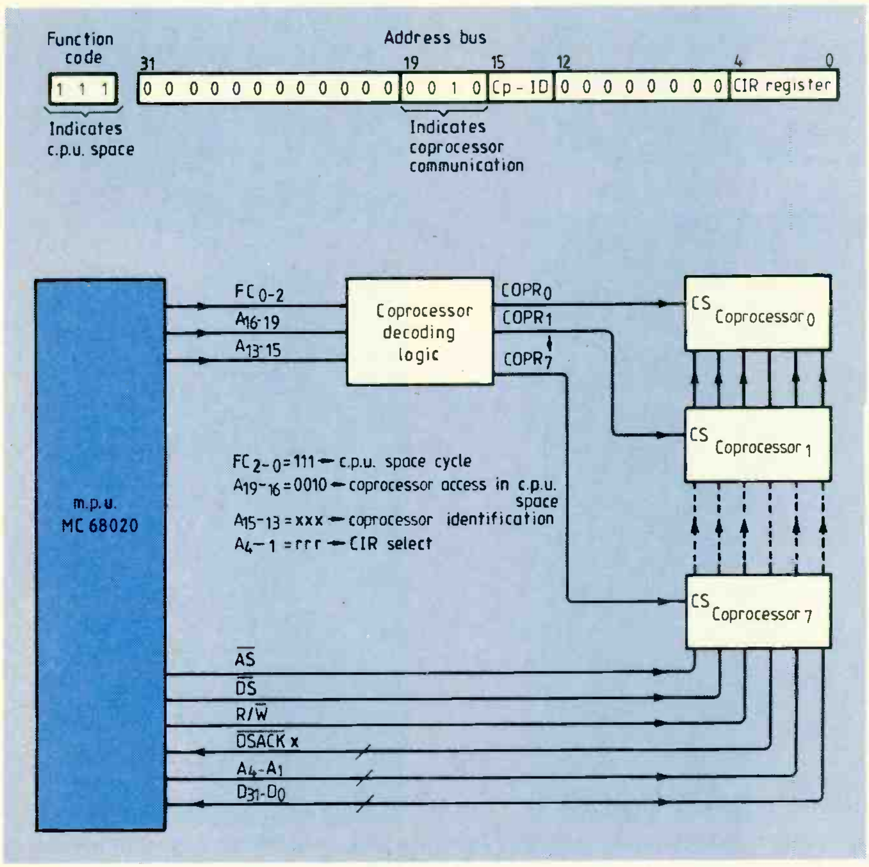
Accessing coprocessors over the coprocessor interface is straightforward since the interface is implemented using standard M68000 asynchronous bus structure without the need for any special signals. This not only makes the interface simple; because of the asychronous nature, the main processor and coprocessor can be operating at different clock frequencies. Designers can therefore optimize a system to make best use of the speed options available.

The coprocessor need not be architecturally similar to the main processor but can be designed so that it best suits its required application. The only requirement is that it adheres to the coprocessor interface protocol. A coprocessor can indeed be implemented as a v.l.s.i. device, as a separate board or even as a separate computer.

When communicating with a coprocessor the MC68020 executes bus cycles in c.p.u. space to access a set of interface registers (CIR). The 68020 indicates that it is accessing c.p.u. space by encoding the function-code lines as all high ($FC_{0\text{-}2} = 111_2$). Chip selection of the coprocessor and the relevant register is then performed by the address bus.

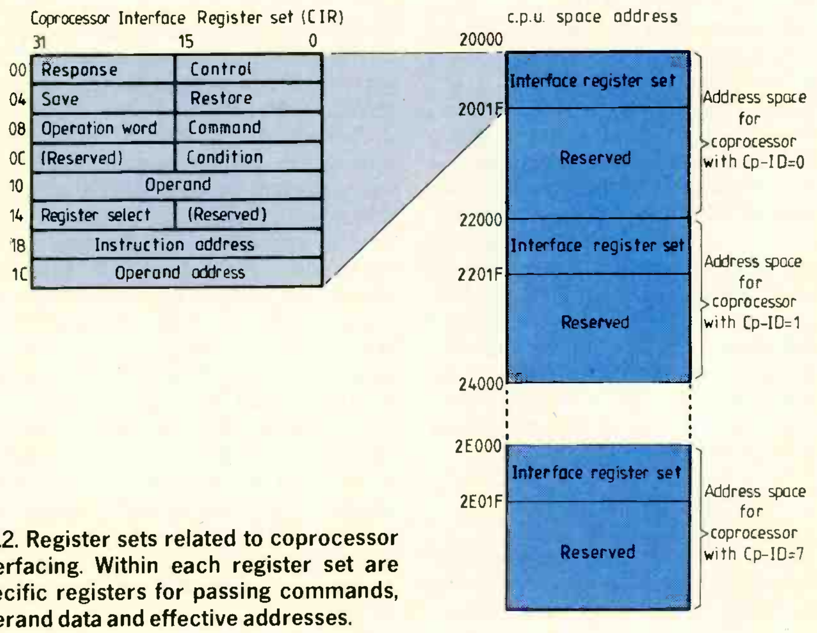
Encoding of the address bus during coprocessor communication is shown in Fig. 1a. This illustration shows that by using the 'Cp-ID' field on the address bus up to eight separate coprocessors can be interfaced concurrently to the MC68020. Figure 1b shows how simply this can be done.

You can see that, if required, there could be several 68881 floating point coprocessors operating concurrently in your system to facilitate very fast and complex number crunching. Interfacing to these separate



Function code
1 1 1
Indicates c.p.u. space
Address bus
31 19 15 12 4 0
0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 1 0 | Cp-ID | 0 0 0 0 0 0 0 0 | CIR register
Indicates coprocessor communication
m.p.u. MC 68020
FC0-2
A16-19
A13-15
Coprocessor decoding logic
COPR0
COPR1
COPR7
CS Coprocessor 0
CS Coprocessor 1
CS Coprocessor 7
FC2-0=111 ← c.p.u. space cycle
A19-16=0010 ← coprocessor access in c.p.u. space
A15-13=xxx ← coprocessor identification
A4-1 =rrr ← CIR select
AS
DS
R/W
DSACK x
A4-A1
D31-D0

Fig.1. Address bus coding during coprocessor communication (a) and an illustration of how up to eight coprocessors can be connected (b) using the Cp ID field.



Coprocessor Interface Register set (CIR)
31 15 0
00 Response | Control
04 Save | Restore
08 Operation word | Command
0C (Reserved) | Condition
10 Operand
14 Register select | (Reserved)
18 Instruction address
1C Operand address
c.p.u. space address
20000
2001F
22000
2201F
24000
2E000
2E01F
Interface register set
Reserved
Address space for coprocessor with Cp-ID=0
Address space for coprocessor with Cp-ID=1
Address space for coprocessor with Cp-ID=7

Fig.2. Register sets related to coprocessor interfacing. Within each register set are specific registers for passing commands, operand data and effective addresses.

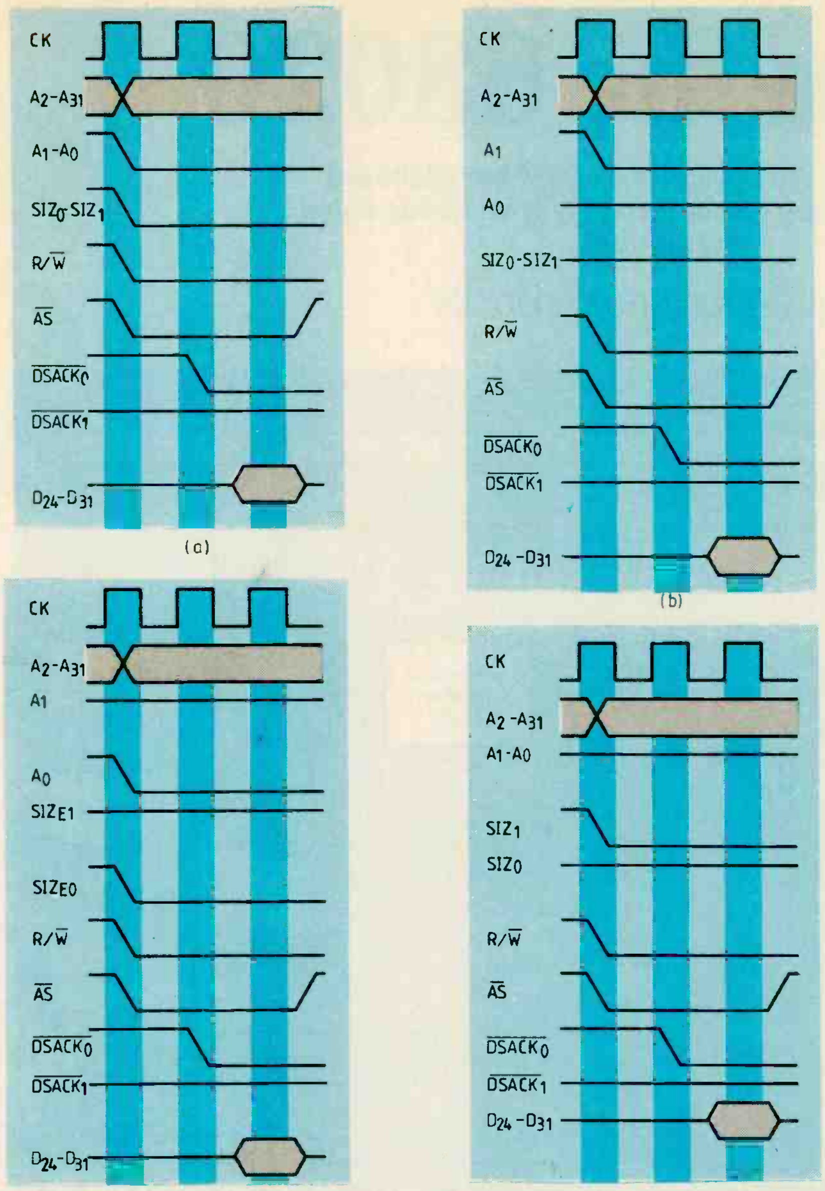


Fig.3. Timing of the four cycles required to write a long word into an eight-bit memory. In all four diagrams, DSACK_0 is zero and DSACK_1 is one, indicating that the memory is eight bits wide. At (a), signals $\text{SIZ}_{0,1}$ are both low to indicate that the processor has four bytes to transfer and the states of these two signals change from 1,1 to 0,1 to 1,0 in (b), (c) and (d) to indicate three, two and one bytes to transfer respectively.

coprocessors is simply a matter of encoding the relevent Cp-ID in the coprocessor instruction, and hence on the c.p.u. space address-bus encoding, so that the MC68020 will communicate with the relevant register set in c.p.u. space.

Figure 2 shows how the separate interface register sets are located in c.p.u. space. Within this interface register set the various registers are allocated to specific functions required for operating the coprocessor interface. There are registers specifically for passing information such as commands, operand data and calculated effective addresses (effective address calculations, and associated operand fetches, are performed by the main processor). Other registers are allocated for use during a context switch when the internal state of the coprocessor needs to be saved and then restored.

DYNAMIC BUS SIZING

The 68020 can dynamically change the size of the data bus on each bus cycle. This feature has been included so that the processor can communicate with peripheral devices intended for 32, 16 or 8-bit buses. Dynamic sizing can also be used to retrofit a 68020 in a 16-bit system and although the full performance increase of a 32-bit processor is not gained, performance improvement can be considerable.

Four signals have been added to support dynamic bus sizing, namely $\overline{\text{DSACK}}_{0,1}$ and $\text{SIZ}_{0,1}$. Data transfer and device-size acknowledge signals $\overline{\text{DSACK}}_{0,1}$ replace the DTACK (data transfer and acknowledge) asynchronous-bus handshaking on the 68000. As with the 68000, these signals are used to terminate the bus cycle but they also indicate the external size of the data bus. Signals $\text{SIZ}_{0,1}$ are outputs indicating how many bytes are still to be transferred during a given bus cycle.

To illustrate this principle, consider the 68020 writing a long data word (32 bits) to an eight-bit memory device. (This would have a serious effect on system performance as four write cycles would be needed). Bus timing diagram Fig.3 shows the four cycles required to write the information into memory.

Using dynamic bus sizing, during a write cycle the 68020 always drives the entire 32-bit data bus even though all 32 bits may not be used. If data is transferred as a byte, it is placed on $\text{D}_{24\text{-}31}$, if it is transferred as a word it is placed on $\text{D}_{16\text{-}31}$ and if it is transferred as a long word $\text{D}_{0\text{-}31}$ are used.

A multiplexer within the 68020 routes data to various sections of the data bus depending on bus size. Address lines $\text{A}_{0,1}$ are linked to this multiplexer. Their encoding indicates a one, two or three-byte offset for a long word to be read from or written to memory.

Unlike other 68000-family processors, the 68020 allows you to place data misaligned in memory, including the user and supervisor stacks. As far as the programmer is concerned this misalignment goes unnoticed in the hardware but it affects performance by increasing the number of data transfers. In fact the only limitation on data storage in the 68020 is that the instruction word, or op-code, must lie on a word or long-word boundary. This is to retain upward software compatibility with 68000-family software.

Consider this example illustrating the principle of misaligned data transfers. Data is transferred to memory by the 68020 over a 32-bit data bus, however the memory address has been offset by one byte from a long-word location.

Figure 4, in which the timing diagrams are simplified to show just the signals used to control data flow, clarifies the situation. Tables summarize decoding of the SIZ and DSACK signals. Also shown is a representation of how the data is organized within the processor.

Since the data is misaligned by one byte the processor needs to make two memory accesses to transfer the long word. During the first cycle, three bytes of data are transferred and the second cycle transfers the last byte.

During the first transfer the processor sets the SIZ pins low to tell the memory that the processor has four data bytes to transfer (a long word). The memory address is odd and offset by one byte from a long-word address so address lines $\text{A}_{0,1}$ are at logic one and zero respectively.

Using $\overline{\text{DSACK}}_{0,1}$, the memory controller indicates that it is 32 bits wide. Information placed on the data bus, in long-word form displaced by eight bits, is carried on $\text{D}_{23\text{-}0}$. The lower eight bits are not transferred during this cycle. Information placed on $\text{D}_{31\text{-}24}$ is just a mirror image of data on $\text{D}_{23\text{-}16}$ and should be ignored by the controller during a write cycle.

During the second transfer, the long-word transfer is completed. Again, $\overline{\text{DSACK}}_{0,1}$ indicate a 32-bit port but the SIZ pins indicate to the memory controller that only one byte remains to be transferred. Data is transferred as one byte on address/data lines $\text{D}_{24\text{-}31}$. Inside the processor, data is transferred within the registers on the opposite end of the data bus. The remainder of the data bus carries a mirror image of this data and should be ignored during write cycles to memory.

Figure 5 shows data transfers over a 16-bit port misaligned by one byte. In this example, it takes three data transfers to transfer a long word. These examples illustrate that the 68020 can be designed into systems with 32, 16 or indeed 8-bit data buses and with a large possibility of data being placed in aligned and misaligned memory.

CACHE MEMORY

Increasing the speed of the 68020 from 16.7MHz to 20 and 25MHz has been accompanied by increases in the cost and difficulty of interfacing the device to external memory without using wait states. The device's internal 256-byte instruction cache relieves this problem.

A minimum of three clock cyles is required when the processor accesses external memory. However if the information is held in the cache, which can be thought of as very fast on-chip local memory, then only two clock cycles are required.

Computer simulation tests were carried out for the 68020 based on the 68000 architecture to find out what type and size of cache would be most beneficial; 256 bytes was found to be the best compromise between efficiency and cost. The cache theory rightly assumes that modern computer programming involves the program repeatedly executing small sections of code as opposed to randomly jumping over large linear address spaces.

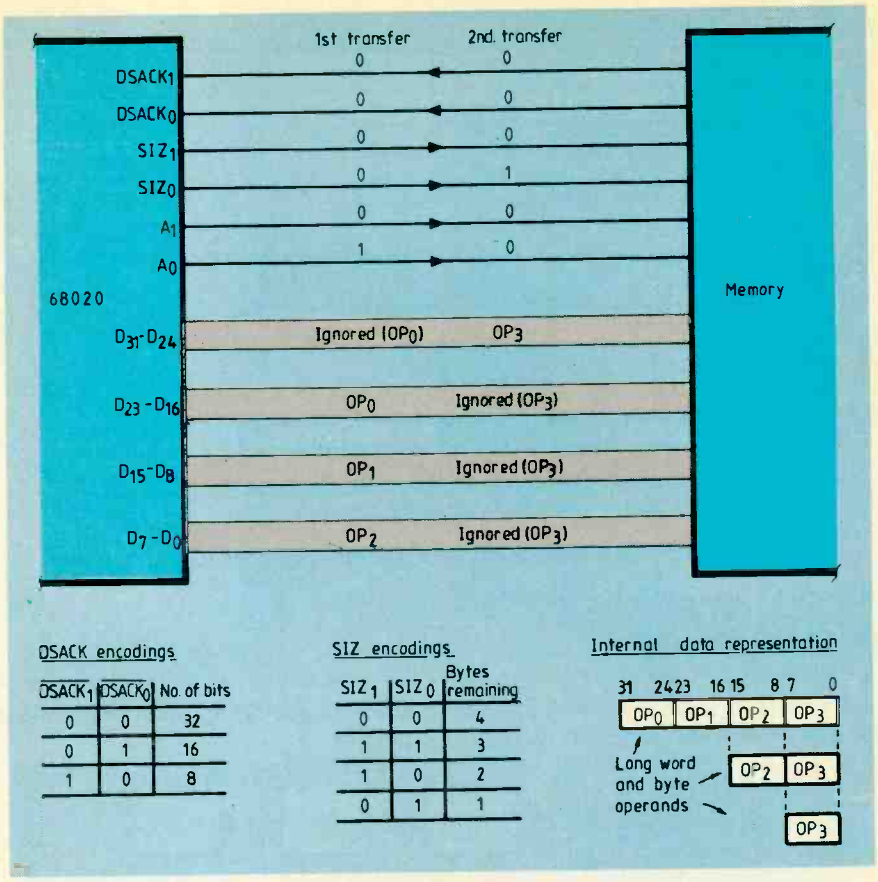
When the processor fetches an instruction from memory, the processor is redundant since no processing can be performed until the instruction has been decoded (this is not strictly true for the 68020). If this instruction fetch can be performed from the cache then the processor spends less time waiting for information from external memory. This has an even greater effect if there is a memory-management unit (m.m.u.), external bus or magnetic backup storage in the system. In such cases it may not always be possible to access external memory with no wait states.

The cache, Fig.6, is 'hit' when address field A_{8-31} and function-code FC_2 (indicating instruction accesses) match the international cache tag field. Cache hit is a term used to describe the condition where the address and any other control information presented on the bus matches information previously placed in the cache tag field.

Some 64 long words are available for storage of cache information. Since the cache is always updated on a long word, maximum throughput is achieved when two instructions in memory are held in the cache. Address lines A_{2-7} select one of 64 entries. Upon reset, the cache is disabled and all entries are made invalid; the v bit in the tag field is also cleared.

Two registers are used with the cache – the cache-control register, CACR, and the cache address register, CAAR. Enabling, disabling and clearing of the cache is carried out by the control register. This register can also be used to freeze individual entries in the cache so critical code sequences can be run within the cache.

It is not possible for the programmer to access cache entries directly. Programming the cache registers is performed using the MOVEC instruction and so can only be done in



	1st transfer	2nd transfer
DSACK1	0	0
DSACK0	0	0
SIZ1	0	0
SIZ0	0	1
A1	0	0
A0	1	0

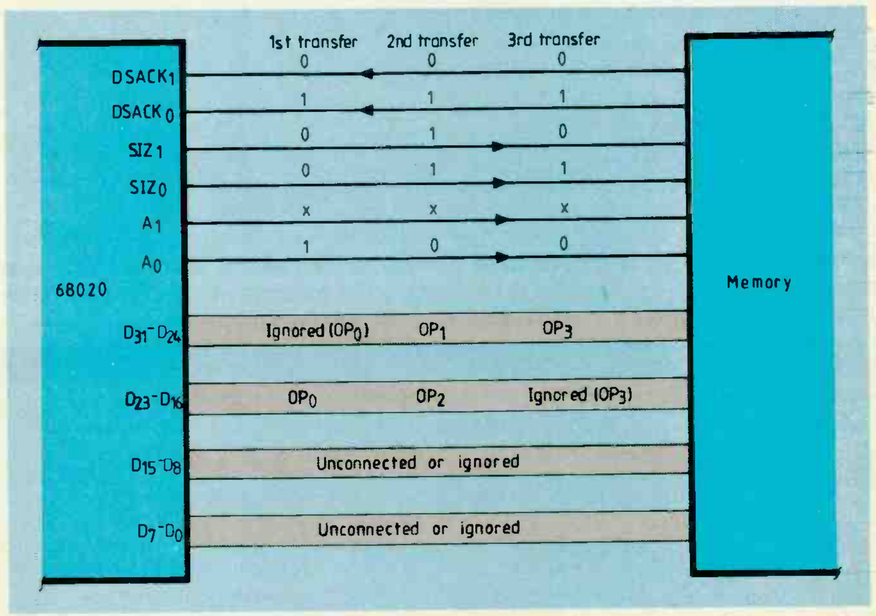
DSACK encodings

DSACK1	DSACK0	No. of bits
0	0	32
0	1	16
1	0	8

SIZ encodings

SIZ1	SIZ0	Bytes remaining
0	0	4
1	1	3
1	0	2
0	1	1

Fig. 4. To send data to memory over a 32-bit address bus when the memory address is offset by one byte from a long-word location requires two transfers. The operand is 32 bits and the boundary is one byte.



	1st transfer	2nd transfer	3rd transfer
DSACK1	0	0	0
DSACK0	1	1	1
SIZ1	0	1	0
SIZ0	0	1	1
A1	x	x	x
A0	1	0	0

Fig.5. To transfer a long word over a 16-bit port misaligned by one byte takes three transactions. The operand is 32 bits and the boundary odd. Data lines D_{0-15} are either unconnected or ignored by both the processor and memory on 16-bit port-sized transfers.

supervisor mode. This ensures that the user cannot accidentally effect the cache operation. Using cache clearing, the operating system can perform a fast context switch in just one instruction.

In addition to the software cache-enable facility, an external hardware cache disable pin, CDIS, can be used to dynamically disable the cache on the next internal cache-access boundary.

THE PIPELINE

Within the 68020 is a three-stage pipeline used for instruction execution, Fig.7. Instructions enter the pipe either from the external bus or from the instruction cache, also within the processor. These instructions are not the ones currently being executed but are 'prefetched' instructions obtained by the bus-interface unit.

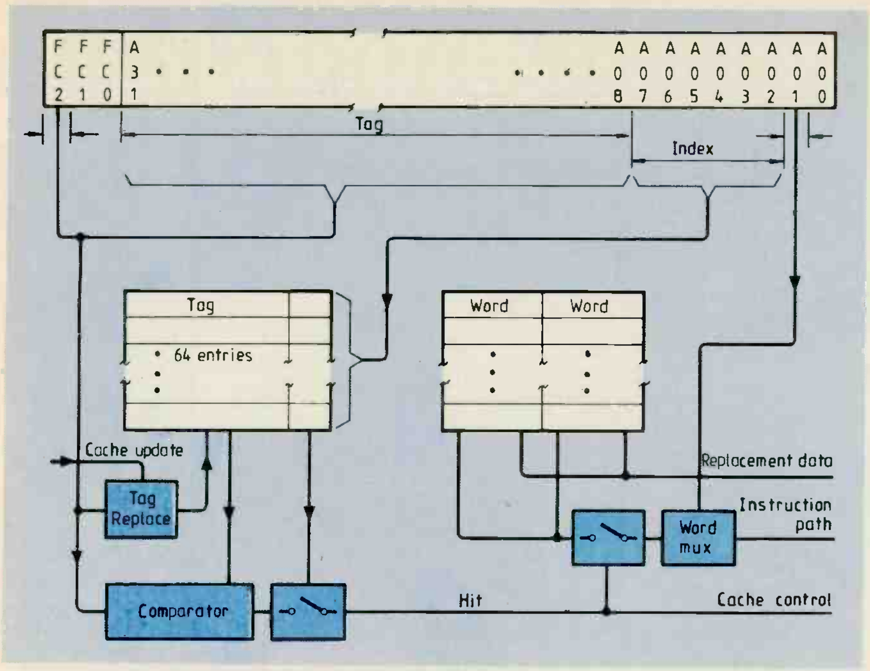


Fig.6. Accessing external memory slows down processing so the 68020 processor has a 256-byte internal cache memory to reduce this problem.

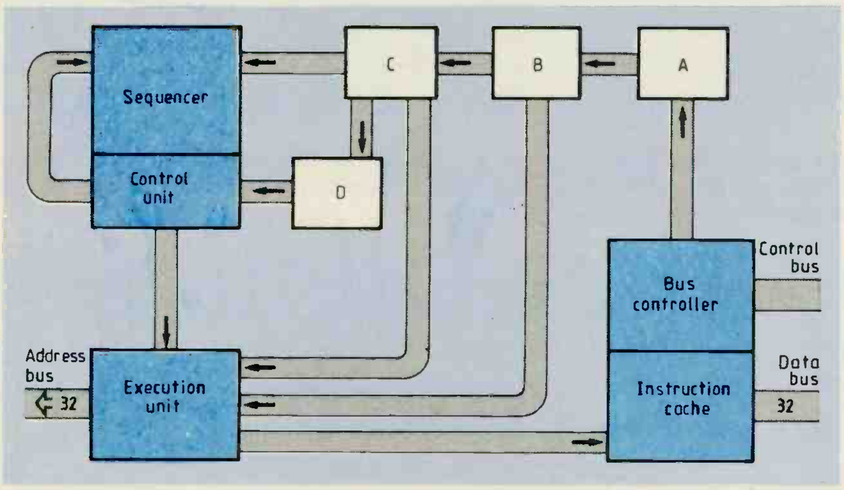


Fig.7. Within the 68020 is a three-stage pipeline which reduces execution time since external bus cycles are not needed to fetch instruction extension words, etc. Instructions enter the pipe either from the external bus or from the instruction cache.

MASTER/INTERRUPT STACKS

Most high-performance microprocessors have two stack pointers. One is usually to the system stack, reserved for interrupts, etc., and the other is the user stack for temporary data storage and parameter passing. In the 68020, these stacks are A_7 and A_7'.

During normal operation most code will be executed in user space and programs will use the A_7 stack for temporary data storage and parameter passing between software routines. Interrupt stack A_7' will only come into use when an exception occurs, such as an external interrupt when control is passed to supervisor mode and the relevant exception processing performed.

In many microprocessors this supervisor stack pointer is the only one accessible during exception processing and all data storage and context switching has to be performed on only one stack. With complex multi-tasking maintaining the main system stack costs processing time; interrupt information for the program counter and status register is interleaved with process-control blocks for various software tasks.

This problem is alleviated in the 68020 by a third stack – the master stack – specifically for holding process-related information for the various tasks. When the master stack is enabled, through bit M in the status register, all non-interrupting exceptions like divide-by-zero, software traps and privilege violation are placed in the user's process-control block on the master stack.

When the first interruption occurs, typically a timer interrupt from a preemptive scheduler, the processor places the program counter, PC, the status register, SR, and the vector offset on the master stack A_7''. It then duplicates this information on the interrupt stack A_7'. The processor is now free to manipulate the processor control block without any further interrupt information being placed on the master stack.

All subsequent interrupts received while performing exception processing are placed only on the interrupt stack A_7'. An effective context switch can now be performing by simply reloading the master stack pointer and mapping in another task's process-control block. This allows context switching to be performed without any master-stack modification by higher-priority interrupts which may occur during the exception processing for the preemptive scheduler's timer interrupt.

With prefetching, long words are always fetched and the cache is arranged as long words. After a prefetch has been requested, the long-word obtained is placed in a 32-bit holding register (stage A) called the cache holding register. This register is used to hold the prefetched long word in the case of a cache miss or to hold the prefetched long word from the cache if a hit occurs.

When a cache hit occurs, the external bus performs an aborted cycle with the $\overline{ECS}$ pin asserted followed by a second assertion of $\overline{ECS}$. Address-strobe pin AS is not asserted during a cache hit. If data is prefetched from the external data bus, it is routed directly to the 32-bit arithmetic logic unit, or a.l.u. This a.l.u., with which is associated a 32-bit barrel shift register, performs arithmetic and logical operations on data registers.

From the cache holding register, instructions pass to stages B, C, and D, where the instruction is executed. Movement through the pipe is governed by the execution time of the instruction at stage D. On reaching stage D of the pipe, the executing instruction corresponds to the program counter. However the processor needs to know where the extension words, etc, for the instruction are so temporary pointers are set up for each stage of the pipe and holding register. These pointers are used to obtain data from each stage of the pipe to allow completion of the instruction.

Using pipeline architecture, the processor operates much faster since no external bus cycles are needed to fetch extension words, etc. Flow of the temporary pointers is tracked by a 32-bit arithmetic unit. There is a third arithmetic unit in the 68020 for arithmetic operations between address registers such as calculating effective addresses. To produce effective addresses needed by the instruction when it is in stage D of the pipe, stage C can be used for effective address calculation. Stages C and D have inputs for allowing them to control the 32-bit a.l.us and a.us.

David Burns graduated with an Honours degree in Electronic and Microprocessor Engineering from Strathcylde University in 1983. Since then he has worked for Motorola Semiconductors in East Kilbride. Presently David is working as a 16/32 bit Applications Engineer and has just completed work on a 25MHz high performance MC68020 computer board.

David's hobbies include orienteering, tennis and badminton.

David Jones graduated with an honours degree in Electronic Engineering at Heriot-Watt university in 1982. Since then he has worked for Motorola Semiconductors in East Kilbride. After working as an equipment engineer David transferred to the microprocessor applications group, progressing from eight to 16 to 32 bit processor design. He is currently working on a 25MHz high performance MC68020 computer board.

David's hobbies include white-water canoeing (both recreational and competitive), swimming and car maintenance. He enjoys outdoor life.

Within the 68020

To make best use of the 68020's upgrading to 32 bits, the 68000-family instruction set has been expanded.

DAVID BURNS AND DAVID JONES

Widening the address and data buses of the 68020 to 32 bits meant that new instructions were needed to make best use of the extra width. For example with an addressing range of up to 4G-byte it became necessary to have branch instructions with 32-bit displacement to reach any address.

Other instructions, like the register-boundary check and signed/unsigned multiply instructions CHK, MULS and MULU, have been extended to operate on 32 bits operands. In addition, the divide instruction DIV now operates on 32 and 64 bits of data and DIVS can be used to perform a 64-by-32-bit division. A new instruction, DIVSL, has been added specifically for 32-by-32-bit long-word division.

Arithmetic capability of the 68020 is improved. To speed up operation of shift and rotate instructions, a 32-bit barrel shift register has been designed into the processor. This allows from 1 to 32 shifts or rotations to be made within a register in only one clock cycle.

Two other instructions for sign extension and linking lists of local data, EXT and LINK, have been modified. Sign extension of a byte to a word or long word is now possible using the modified EXT instruction EXTB, and LINK can accommodate a 32-bit displacement.

Lengths of some instructions executed on the 68020 may differ from the same instructions executed on the 68000. This is because the new addressing modes require an additional 32-bit instruction-extension word which is decoded by the processor.

INSTRUCTION ENHANCEMENTS

New instructions have been added to give additional flexibility for the programmer. One of these is CHK2 which performs an upper and lower boundary check on a register's content, instead of just on the upper boundary as with other 68000 processors.

With the 68000, the TRAPV instruction causes a system trap which routes to the operating system via a supervisor access if the condition-code register overflow bit is set. This instruction modified to $\mathrm{TRAP_{CC}}$ now responds to all the bits in the condition-code register.

In addition to the coprocessor general function instructions for passing command words to coprocessors such as the MC68881 floating-point device, there are six instructions for testing or controlling active coprocessors within a 68020 system. Their mnemonics are cpSAVE, cpRESTORE, $\mathrm{cpB_{CC}}$, $\mathrm{cpDB_{CC}}$, $\mathrm{cpS_{CC}}$ and $\mathrm{cpTRAP_{CC}}$.

The last four of these instructions operate in the same way as 68020 instructions $\mathrm{B_{CC}}$, $\mathrm{DB_{CC}}$, $\mathrm{S_{CC}}$ and $\mathrm{TRAP_{CC}}$ except that they operate on the coprocessor condition-code register. During assembly, these instructions are given an identifcation field (ID) corresponding to one of eight possible coprocessors. Using these test and control instructions, the programmer can make judgements based upon the results of a coprocessor operation.

Instructions cpSAVE and cpRESTORE perform a context switch on a processor. Each time a context switch occurs (the operating system switches to a new user or runs a new task) all internal information that the coprocessor requires to perform instructions is placed on the supervisor stack. After the context switch the new user or task has full use of the coprocessor internal programming registers. To restore the coprocessor to its original condition, i.e. its state before the cpSAVE instruction was issued, cpRESTORE is used.

SUPERVISOR INSTRUCTIONS

Improvements have been made to the 68020 supervisor mode. A number of specific instructions belong to this group. To accommodate programming of the 68020's additional control register the MOVEC instruction now covers cache control and address registers.

For example a MOVEC.L D0,CACR operation loads the value contained in D0 into the cache control register. The value represented by D0 can indicate a cache enable, a cache clear, a cache freeze or a clear entry operation.

To perform the debugging task of inserting breakpoints into a code sequence the 68000 processor needs to execute a predefined illegal instruction. On receiving such an illegal instruction the processor jumps to the illegal instruction's exception-handling routine and executes the required breakpoint task. In a monitor program for example it may well halt the program and display the internal register contents on a terminal.

Using this method, the processor has to store an entry for the displaced code in some form of breakpoint table in memory. A large amount of software is required to manage this table, for example to replace the original opcode after the breakpoint has been finished with by the programmer.

A dedicated breakpoint instruction, BKPT, is included in the 68020 and used in the form BKPT#<data>. Since the immediate data value represented by#<data> can be from 0 to 7, up to eight hardware breakpoints are possible. In Motorola assembly language, the # sign designates an immediate value.

On executing a breakpoint instruction the 68020 reads a word from the c.p.u.-space address corresponding to the breakpoint number, Fig.1. If the $\mathrm{DSACK_x}$ signals terminate the cycle then the data at this address is latched into the processor. This data is the 16-bit opcode that was displaced to make room in memory for the breakpoint instruction. If this breakpoint acknowledge cycle is terminated by the BERR control signal then the processor performs exception processing for an illegal instruction – as with the 68000.

This hardware process is much faster than the 68000 equivalent since the breakpoint table is managed by the processor and not the system software. The sequence of events illustrated in Fig.2 shows that when a breakpoint instruction is placed in a code sequence (by a monitor program, say) it places the displaced opcode into its corresponding c.p.u.-space register or memory location.

The remaining 16-bits of this register can be used to implement a breakpoint count. This count is loaded with the number of times the program should execute the displaced instruction before the breakpoint is actually taken. For example if BKPT is executed and the count is non-zero then the external hardware will need to generate

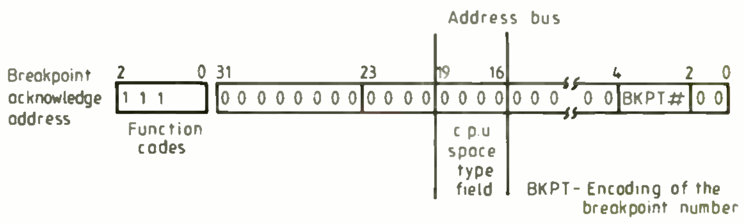


Fig.1. During a breakpoint-acknowledge cycle the 68020 addresses this memory location in c.p.u. space (function codes all ones) to fetch the displaced opcode. If the breakpoint is to be taken then the memory access should terminate in a BERR, otherwise $\mathrm{DSACK_x}$, are asserted.

Two previous articles presented in our December and January issues introduced the 68000 family and discussed 68020 architecture.

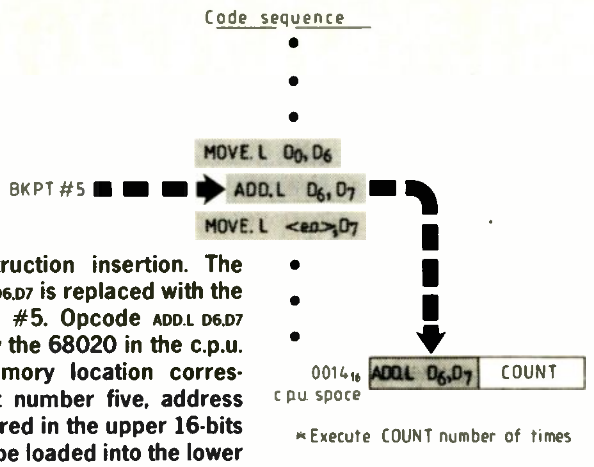


Fig.2. Breakpoint instruction insertion. The 16-bit opcode for ADD.L D6,D7 is replaced with the 16-bit opcode for BKPT #5. Opcode ADD.L D6,D7 must then be stored by the 68020 in the c.p.u. long-word (32-bit) memory location corresponding to breakpoint number five, address 14_{16}. The opcode is stored in the upper 16-bits and a count value can be loaded into the lower 16-bits.

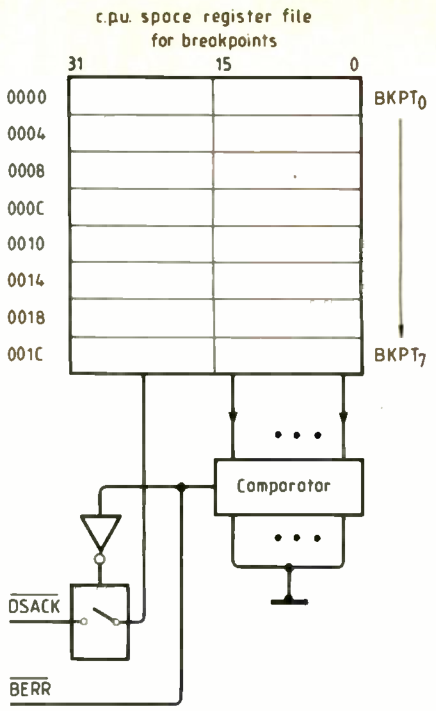


Fig.3. External breakpoint hardware. Eight long-word memory locations from $0\text{-}1F_{16}$ in c.p.u. space are used to temporarily store the opcode replaced by a BKPT instruction. Each time BKPT is executed the 68020 accesses one of the locations depending on which breakpoint is being executed ($BKPT_0 - BKPT_1$) If the count value (bits 0-15) is not zero the replaced opcode is placed on the data bus (bits 16-31) and the $\overline{DSACK}_x$ lines are asserted. If the count is zero then $\overline{BERR}$ is asserted to initiate a breakpoint acknowledged cycle in the 68020.

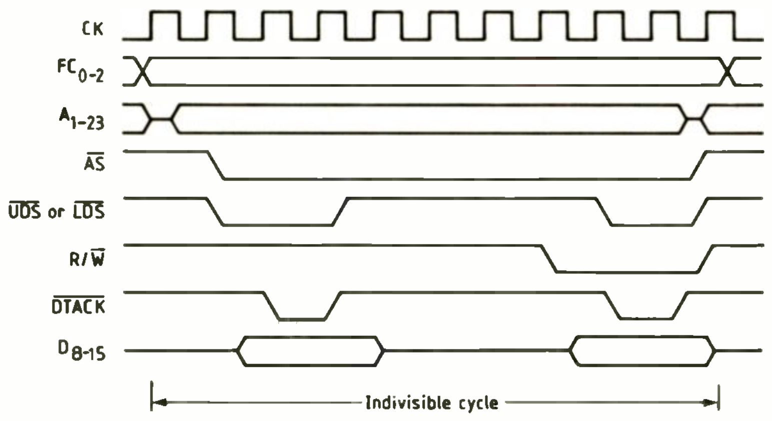


Fig.4. When executing the TAS instruction (test and set an operand) the cycle is made indivisible, i.e. uninterruptible, by continued assertion of address strobe $\overline{AS}$ through the read and write operations.

$\overline{DSACK}_x$ in order to execute the displaced code. When the count reaches zero the hardware should generate $\overline{BERR}$ and exception processing will begin for the breakpoint service routine. For simplicity, design of this hardware can be incorporated into a memory-management unit as is the case with the MC68851 paged-memory-management unit.

Hardware is shown in Fig.3.

MULTIPROCESSING AND MULTITASKING

The MC68020 has several facilities for multiprocessing. It has special instructions like TAS, CAS and CAS2 for interprocessor or intertask communications and a hardware line ($\overline{RMC}$) for bus locking. An instruction-continuation facility allows easy transfer from one processor to another.

Execution of the special instructions forces assertion of the $\overline{RMC}$ signal (read-modify-write cycle) and thus causes the bus to lock. Any alternative bus masters in the system must wait until the negation of the $\overline{RMC}$ signal before they can take control of the bus.

Instruction TAS (test and set an operand) is the same as that on the 68000 processor and allows implementation of flag variables for globally-shared memory blocks. This instruction allows the testing and setting of a variable to be performed in an indivisible cycle.

With the 68000 this indivisible cycle is achieved by keeping the address-strobe signal ($\overline{AS}$) asserted throughout the read-modify-write operation. You can see this in Fig.4 where the processor reads a memory location, tests the data item and may modify it and then write it back out to memory. This cycle is carried out without negation of the address strobe. With the 68020, execution of this type of cycle is further indicated by assertion of the $\overline{RMC}$ signal.

Flags are very important elements in the implementation of any multitasking/multiprocessor system. In these types of application flags are used to control access to globally-shared memory blocks. If a task or processor needs to gain access to one of these memory blocks it must test the associated flag to determine whether the block is presently being used. If the flag is clear then the task (processor) will claim control of that memory block by setting that flag. It should then be guaranteed sole access to that particular block.

If your system is configured such that reading and testing of the flag is implemented through one instruction and setting of the flag is implemented through another instruction then it is possible that a number of tasks (processors) may gain access to the memory block at one time.

This situation can arise if task A for example reads the flag and, finding it clear, decides to set it and thus claim access to the block. Task B may then interrupt task A (in a multitasking environment there may well be a preemptive-timer interrupt) and it too may read the flag associated with the memory block and also find it clear because task A did not finish writing the data. Task B claims the memory block by setting the flag.

The preemptive interrupt may occur again and reconstitute task A. Execution of task A continues from where it detected the flag as being 'clear and continue' by setting the flag to claim the memory block.

In this way, two tasks concurrently execute the same memory block and as a result there is a risk of data corruption. To remove this possibility, testing and setting of the flag must be performed in an indivisible cycle, i.e. with no interrupts or bus arbitration, which is done using the TAS instruction.

This read-modify-write class of instruction is further expanded by CAS and CAS2. Compare-and-swap-with-operand, CAS, is an extension of the TAS instruction and allowing data items (byte, word or long word) to be compared and swapped. This instruction could for example be used in the manipulation of linked lists when a new item is required to be inserted for example.

A copy of the old starting pointer must first be put at the base of the new item block and then the pointer to the new item block must be updated into the starting pointer area. To perform this reliably, updating of the new starting pointer must be executed using an indivisible cycle.

The first operation is to place a copy of the old starting pointer at the base of the new linked items block. This new item is starting address must now also become the starting address of the linked list. If after placing the old starting pointer at the base of the new item the present process, task A, was interrupted then task B could intervene, placing its own new item in the linked list.

The pointer at the base of task A's item is no longer the next item in the linked list – it doesn't know of the existence of task B's item. To remedy this problem it is necessary to retest the start pointer before you update it.

This can simply be done by using the instruction CAS DC, DU, START. To give an example, the existing starting pointer is copied into data register DC and then by using the CAS instruction the value in this register (old starting pointer) is compared with the present starting pointer (START). If these two are the same then the starting pointer is updated with the value in the other data register, D_U, (the new starting pointer). If the values in data register, D_C, and START don't compare (i.e. the START pointer has subsequently been changed) then the update is not performed and integrity of the linked list is preserved. This function is again performed as an indivisible operation and its bus activity is shown in Fig.5.

Instruction CAS2 is identical to CAS except that it can be used to compare and update two operands within the same indivisible cycle. This is of use in maintaining doubly-linked lists i.e. items with both a next-item pointer and a last-item pointer.

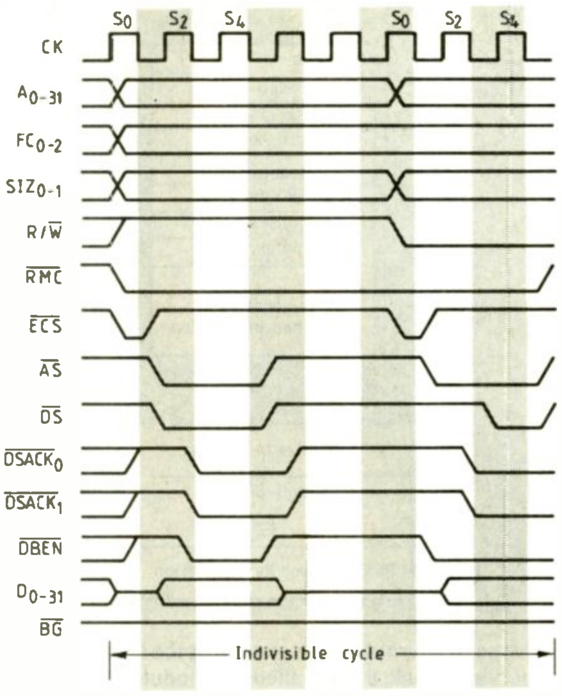


Fig.5. Execution of the CAS instruction (compare and swap with operand) requires a change of address and so continuous assertion of address strobe AS cannot be used. The RMC (read-modify-write cycle) signal indicates indivisible cycles.

MODULE SUPPORT

In comparison with the 68000, the 68020 has more levels of access control than just the supervisor/user split as on previous M68000 processors. Two instructions CALLM and RTM (call module and return from module) can be thought of as advanced subroutines used to gain access to other levels of security.

This feature enhances the computer operating system by allowing a number of layers or shells to be created around the computer kernel. With this type of system the 68020 can monitor attempts by the user to gain access to higher levels of security than permitted. In this case an access level exception will be taken by the processor and an error condition will be flagged. Figure 6 shows what such a system could look like in an operating system application.

In modern computing there are many areas where this mechanism could be used to give more security to an operating system. For example, consider an applications programmer who would like to access a large database to obtain personal details about an employee. For this example the programmer would need to have access permission to obtain this information. In addition the programmer needs to be prevented from updating that information, this being a function of the database manager or system administrator. Another area of use could be in the Unix operating system to set up different shells. Typically Unix runs with a Bourne shell, however other layers or shells are available for Unix; in addition user generated shells can be created.

The 68020 can be used to achieve these features by a combination of software and external hardware. The execution of the CALLM instruction is

CALLM #<data>,<e.a.>

This instruction is very similar to JSR (jump to subroutine), except that the immediate data value is the number of bytes of argument to be passed via the stack to the called module. Effective address <e.a.> is the actual address of the module descriptor in memory.

The module descriptor can be thought of as a gateway through which the calling program must gain access. The module being called can be thought of as a subroutine that is to act upon the data arguments passed to it. This module can be the same priority level or different to that of the caller. Figure 7 illustrates the procedure that the MC68020 goes through during execution of the CALLM instruction.

The CALLM instruction acts as follows. The effective address (<e.a.>) in the instruction points to the address of the module descriptor or gate to find control information. At the same time, the MC68020 creates a module-stack frame similar to an interrupt. The module descriptor contains the following information.

- How arguments are to be passed to the called module (option field). They can be passed either under the module stack or through an indirect pointer in the stack of the calling module.
- Descriptor type (type field). Two types are used; one in which there is no change in access rights (where the module stack is

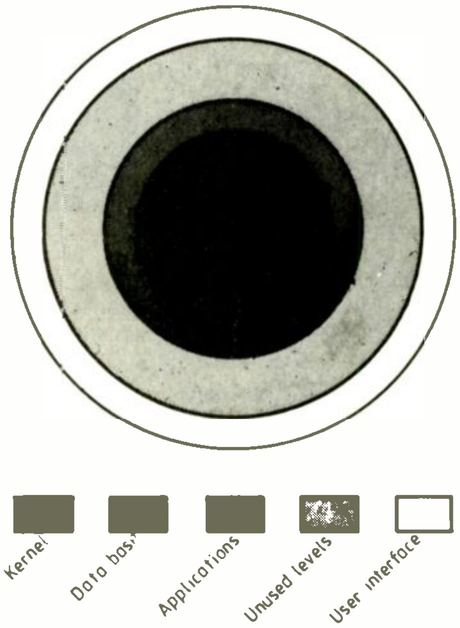


Fig.6. A typical computer can be thought of as layers of protected software programs. At the heart of the computer is the kernel. This is code written such that it will interface directly to the computer hardware. This will typically be the highest priority code and, as such, runs in supervisor mode. Other layers that need protecting from the user programs include database and applications software, these being accessible only to users with high access-permission privileges, e.g. system administrators.

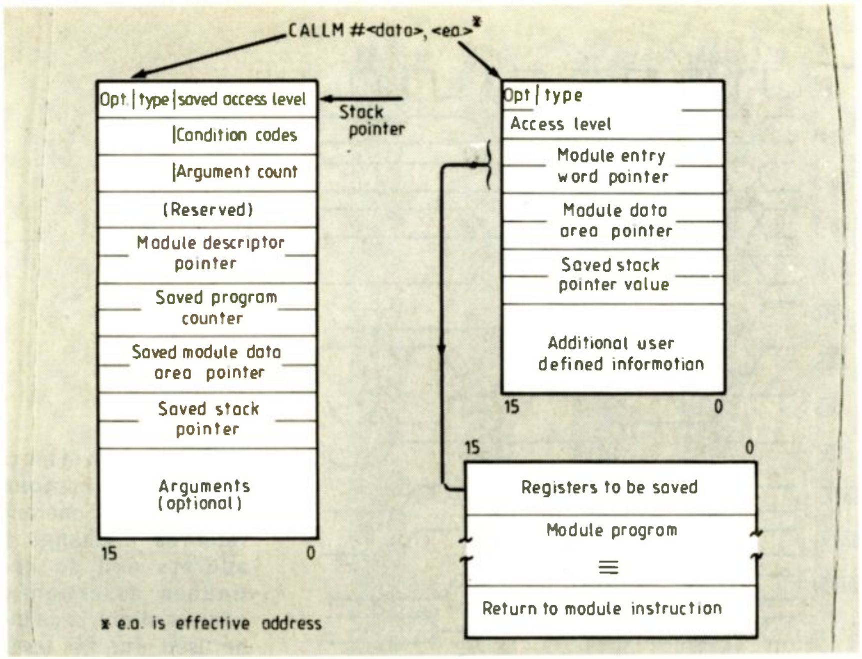
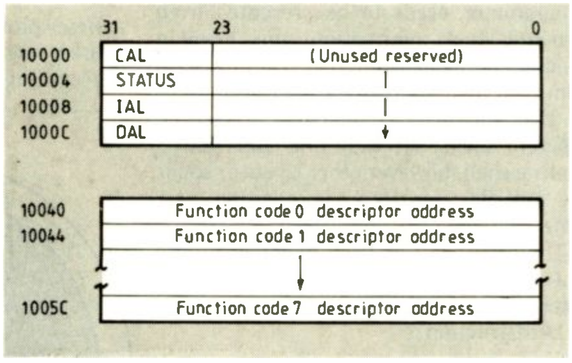

CALLM #<data>, <ea>*
Opt | type | saved access level
Condition codes
Argument count
(Reserved)
Module descriptor pointer
Saved program counter
Saved module data area pointer
Saved stack pointer
Arguments (optional)
15
0
Stack pointer
Opt | type
Access level
Module entry word pointer
Module data area pointer
Saved stack pointer value
Additional user defined information
15
0
15
0
Registers to be saved
Module program
Return to module instruction
* e.a. is effective address

Fig.7. CALLM can be thought of as an advanced subroutine. On executing the instruction the 68020 places information on the supervisor stack; this is called the module stack frame. The effective address given in the instruction (<e.a.>) points to the address in memory of a module descriptor containing specific information about the module being called. Within the module descriptor is the actual address of the program to be executed.

Fig.8. Implementing access-level support requires external hardware at these addresses in c.p.u. memory. Four long words are used to control access privilege and eight long words are used to contain addresses of up to eight module descriptors.


31
23
0
10000
10004
10008
1000C
CAL
STATUS
IAL
DAL
(Unused reserved)
10040
10044
1005C
Function code 0 descriptor address
Function code 1 descriptor address
Function code 7 descriptor address

created on the calling module) and the other where there is a change in access levels (a change of stack pointer may be needed).

- Called module access level.
- Stack address of the called module. Note that the first word of the module program indicates the registers used in the program; this is interpreted by the 68020 in such a way that the processor knows which registers to place on the stack.
- If the access change requires a change of stack pointer, the old value is saved in the module and all the arguments are copied to the new stack.
- User-defined information, for example a count of the number of times the module has been accessed within a certain task.

The module stack frame contains information needed by the processor to continue program execution from the instruction following the CALLM. Arguments passed to and from the module are placed on the stack before execution of the call module instruction. If there is a change in the stack pointer requested from the module descriptor, this is requested and the arguments are copied onto the new stack, typically for access-level changes.

This does not offer a great deal of protection for this system, since the user can obtain access to a higher-privilege level simply by reading the module address in the descriptor. For this reason, the protection mechanism is designed into the 68020 as one of the special c.p.u. functions. Any request for change of access level causes the processor to execute a c.p.u.-space cycle. While processing a type #01 (a module call for which there is a change in stack pointers) descriptor the CALLM and RTM instructions communicate with external access-control hardware in c.p.u. address space at the address shown in Fig.8. These registers would typically be designed into a memory-management unit as an aid to the protection mechanism (as with the 68851 paged memory management unit). As the communication to these registers is performed by the c.p.u. microcode, the user is unaware of these operational checks. The current access-level register (CAL) contains the access level of the currently executing module. The increase access-level register (IAL) is the register through which the calling module requests increased access rights and the decrease access-level register (DAL) is the register through which the processor requests decreased access right. These registers are updated from the module descriptor stack frame.

The access-status register allows the processor to determine from the external hardware whether an intended access-level transition is valid. During the CALLM instruction, the processor uses the descriptor register to communicate the address of the type #01 or descriptor.

The RTM instruction is executed as the last instruction of the called module and is used to restore the original information to the processor and to the next instruction to be executed. After the RTM instruction has executed, resultant arguments will be resident on the stack for the program to use.

Support for up to eight levels of access can now be supported, as opposed to the previous user/supervisor mode on other M68000-family processors.

David Burns and David Jones are with Motorola in East Kilbride.

Futurebus group starts up

In response to the growing number of requests for product and information on Futurebus, the IEEE's 32-bit bus standard, a number of UK companies have established the Futurebus Manufacturers and Users Group. It is the intention of the group to give the UK an early start and a competitive edge in Futurebus products, an opportunity that has been sadly missed with other buses, notably VME.

Twenty invited representatives of 12 UK companies attended the inaugural meeting held last autumn, hosted by National Semiconductor at Swindon. Nat Semi said they were the only manufacturer with Futurebus devices in production, offering a range of transceivers, drivers and receivers which conform to the P896 specification.

Also present and with product available was BICC-Vero who have backpanels, termination networks, extender cards and prototyping boards in current production. A number of other companies are known to have Futurebus products under development including an advanced multiprocessor system based on multiple 32332's, graphics engines and systems for high-speed parallel processing, but most of these products are firmly 'under wraps', with the intending suppliers playing their cards close to their chests. Apart from National Semiconductor, at least four other manufacturers are known to have l.s.i. products for Futurebus currently under development.

The potential market for Futurebus is said to be vast, with very little competition in the 32-bit stakes. It is claimed that it has all the answers to the 'bus driving problem', cache coherency, bus arbitration and bandwidth whilst also being manufacturer and technology independant. (news, Jan 1984, page 45).

Represented at the inaugural meeting National Semiconductor, BICC-Vero, Dean Microsystems, Plessey, British Telecom, Ferranti, Array Consultants, Fraser Williams Industrial Systems, Spectra-Tek and the DTI.