Exhibit N

United States Patent [19]

Edwards et al.

[11] **Patent Number: 4,680,698**

[45] **Date of Patent: Jul. 14, 1987**

[54] **HIGH DENSITY ROM IN SEPARATE ISOLATION WELL ON SINGLE WITH CHIP**

[75] Inventors: **Jonathan Edwards**, Bristol; **David L. Waller**, Avon; **Michael D. May**, Bristol, all of England

[73] Assignee: **Inmos Limited**, Bristol, England

[21] Appl. No.: **552,601**

[22] Filed: **Nov. 16, 1983**

[30] **Foreign Application Priority Data**

Nov. 26, 1982 [GB] United Kingdom 8233733

[51] **Int. Cl.**[4] **G06F 13/00**; G06F 15/16; G06F 7/48; H01L 27/02

[52] **U.S. Cl.** **364/200**; 364/712; 357/40

[58] **Field of Search** ... 364/200 MS File, 900 MS File, 364/700, 706, 712; 365/200; 371/8, 10; 361/400; 357/40, 45

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,980,992	9/1976	Levy et al.	364/200
3,993,934	11/1976	Baker et al.	361/400
4,074,293	2/1978	Kravitz	357/40
4,144,563	3/1979	Heuer et al.	364/200
4,153,933	5/1979	Blume, Jr. et al.	364/200
4,191,996	3/1980	Chesley	364/200
4,346,459	0/0000	Sud et al. .	
4,349,870	9/1982	Shaw et al.	364/200
4,467,420	8/1984	Murakami et al.	364/200
4,482,950	11/1984	Dshkhunian et al.	364/200
4,491,907	1/1985	Koeppen et al.	364/200
4,546,454	10/1985	Gupta et al.	371/10 X

OTHER PUBLICATIONS

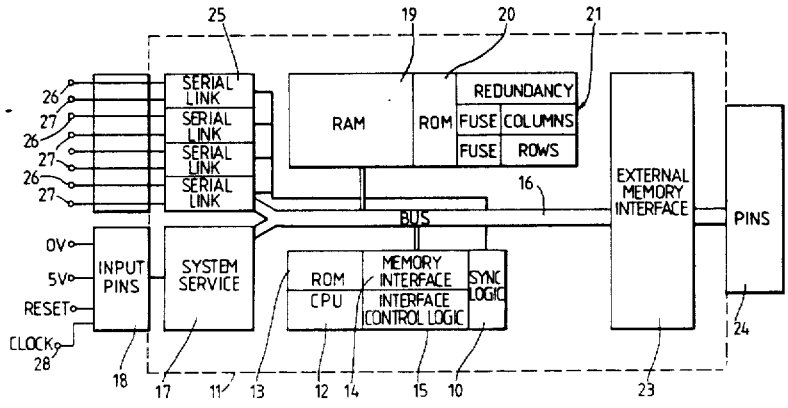
Intel Microcomputer Handbook, Jan. 1983, pp. 16–26.
Electronic Design, Oct. 14, 1982, pp. 131–139.
Electronic Design News, Oct. 27, 1982, p. 165.
Barron, "The Transputer," *The Microprocessor and Its Application*, pp. 343–357 (1978).
"Growing Microfamilies Show Off New Strengths," *Electronics Design*, vol. 29, No. 1, pp. 64–68 (1981).
Tominage, et al., "High Performance 3 Micrometer Memories," *Electronic Engineering*, vol. 54, No. 663, pp. 101–109 (Mar. 1982).
Ohzone, et al., "An 8K×8 Bit Static MOS RAM Fabricated by n-MOS/n-Wel CMOS Technology," *IEEE Journal of Solid State, Circuits*, vol. SC-15, No. 5, pp. 854–861 (1980).

Primary Examiner—Archie E. Williams, Jr.
Assistant Examiner—William G. Niessen
Attorney, Agent, or Firm—Edward D. Manzo

[57] **ABSTRACT**

A programmable, high speed, single chip microcomputer includes 4K of RAM, ROM, registers and an ALU. Program can be stored in the on-chip RAM. The first local variable of each process to be executed is a workspace pointer (WPTR), and each process has a respective workspace identified by its WPTR. For each process, addressing of other variables is relative to the current WPTR, which is stored in a workpiece pointer register (WPTR REG). Instructions are constant bit size, having a function portion and a data portion loaded, respectively, into an instruction buffer (IB) and an operand register (OREGTR). Memory address locations are formed by combining the contents of the workspace pointer register and the operand register, or the contents of the A Register and the operand register. A set of "direct functions" obtains data from OREG. "Indirect functions" use the OREG contents to identify other functions, obtaining data from registers other than the operand register. A "prefixing" function (PFIX) develops operands having long bit lengths. Scheduling and descheduling of processes are achieved by forming a linked list within the several workspaces for the active processes. Each workspace identifies the workspace pointer of the next process to be executed. Each workspace contains in memory the identification of the next instruction to be executed for that respective process. A "last pointer" register (LPTR REG.) cooperates in the scheduling operations. Each microcomputer chip can be coupled serially to other such chips on a respective pair of only two wires, each a unidirectional channel. Each channel also has two registers, one for process identification and one for data. Communications are synchronized.

15 Claims, 19 Drawing Figures



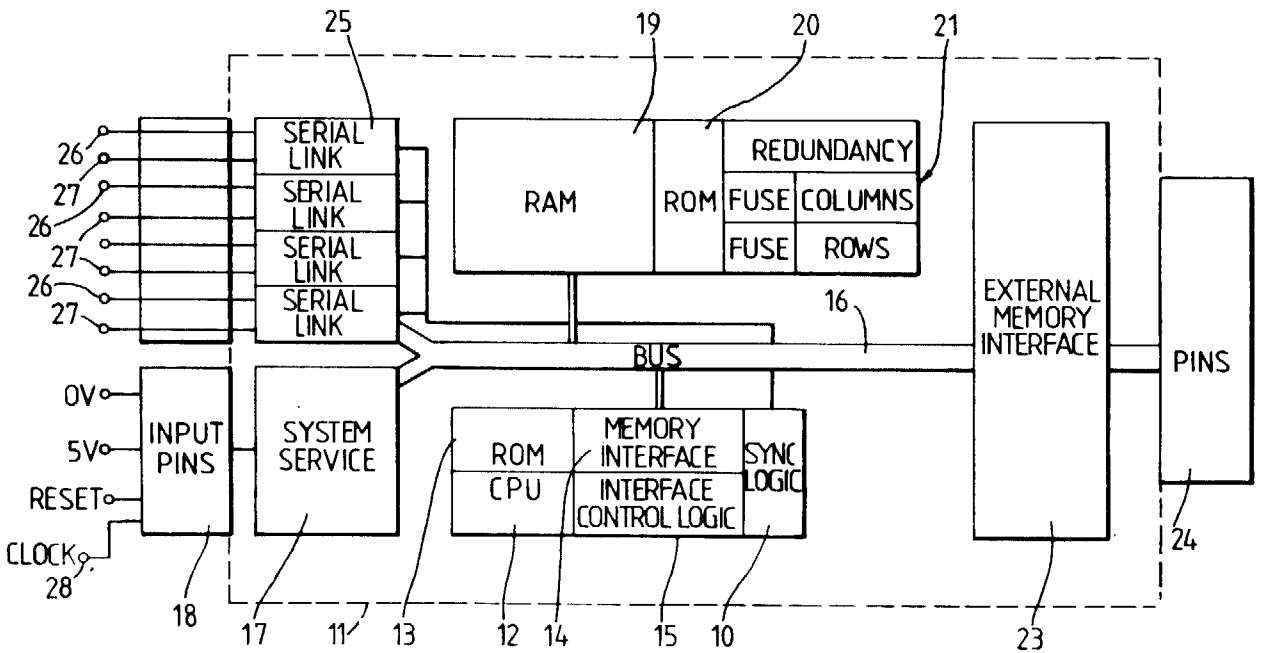

25
19
20
21
SERIAL LINK
SERIAL LINK
SERIAL LINK
SERIAL LINK
26
27
26
27
26
27
RAM
ROM
REDUNDANCY
FUSE
COLUMNS
FUSE
ROWS
16
BUS
EXTERNAL MEMORY INTERFACE
PINS
24
0V
5V
RESET
CLOCK
28
INPUT PINS
SYSTEM SERVICE
ROM
CPU
MEMORY INTERFACE
INTERFACE CONTROL LOGIC
SYNC LOGIC
18
17
11
13
12
14
15
10
23

Fig.1.

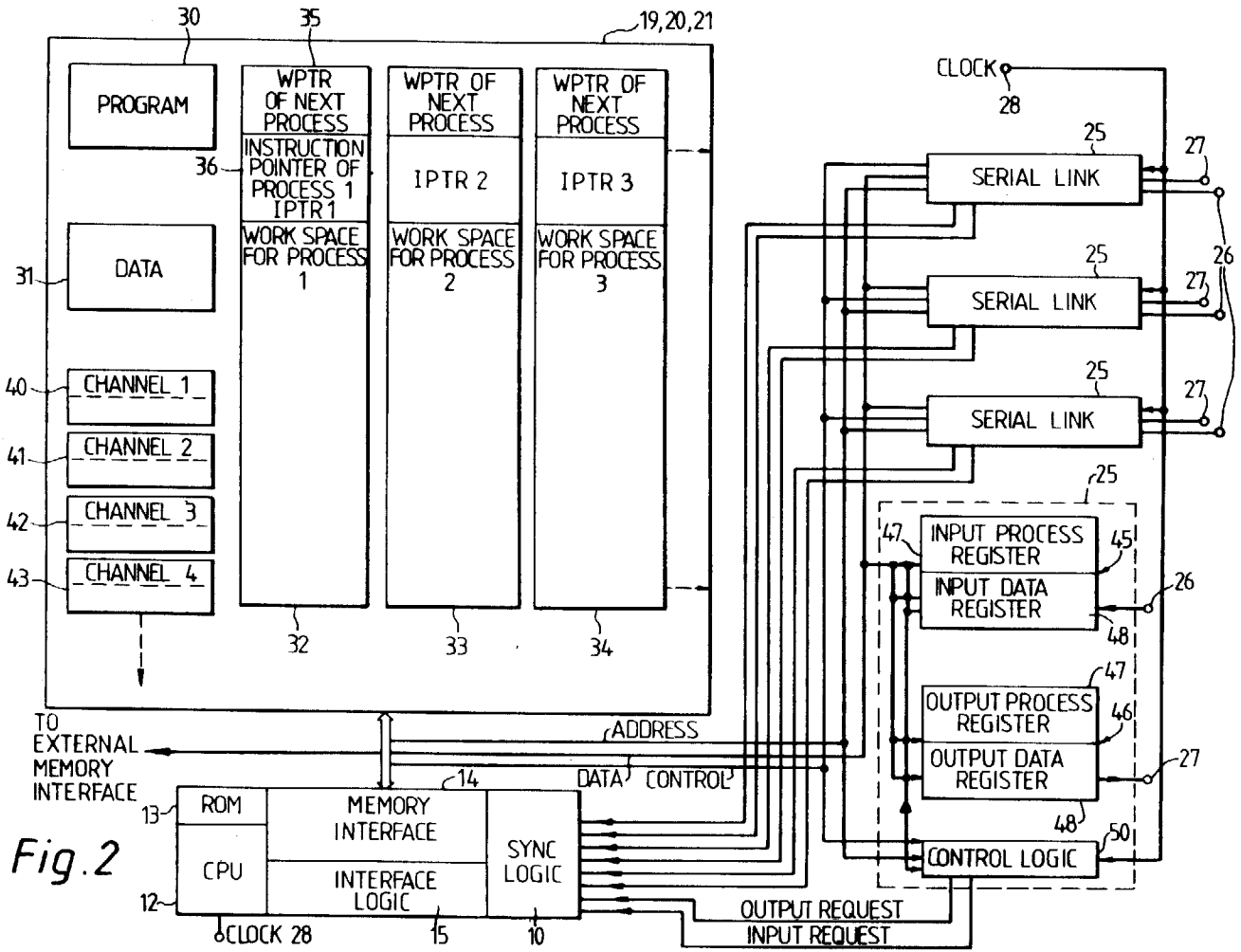

30
35
19,20,21
PROGRAM
WPTR OF NEXT PROCESS
WPTR OF NEXT PROCESS
WPTR OF NEXT PROCESS
36
INSTRUCTION POINTER OF PROCESS 1 IPTR1
IPTR 2
IPTR 3
DATA
WORK SPACE FOR PROCESS 1
WORK SPACE FOR PROCESS 2
WORK SPACE FOR PROCESS 3
31
CHANNEL 1
CHANNEL 2
CHANNEL 3
CHANNEL 4
40
41
42
43
32
33
34
CLOCK
28
25
SERIAL LINK
SERIAL LINK
SERIAL LINK
27
26
INPUT PROCESS REGISTER
INPUT DATA REGISTER
OUTPUT PROCESS REGISTER
OUTPUT DATA REGISTER
CONTROL LOGIC
45
46
47
48
50
TO EXTERNAL MEMORY INTERFACE
ADDRESS
DATA
CONTROL
14
13
ROM
CPU
MEMORY INTERFACE
INTERFACE LOGIC
SYNC LOGIC
12
CLOCK 28
15
10
OUTPUT REQUEST
INPUT REQUEST

Fig. 2

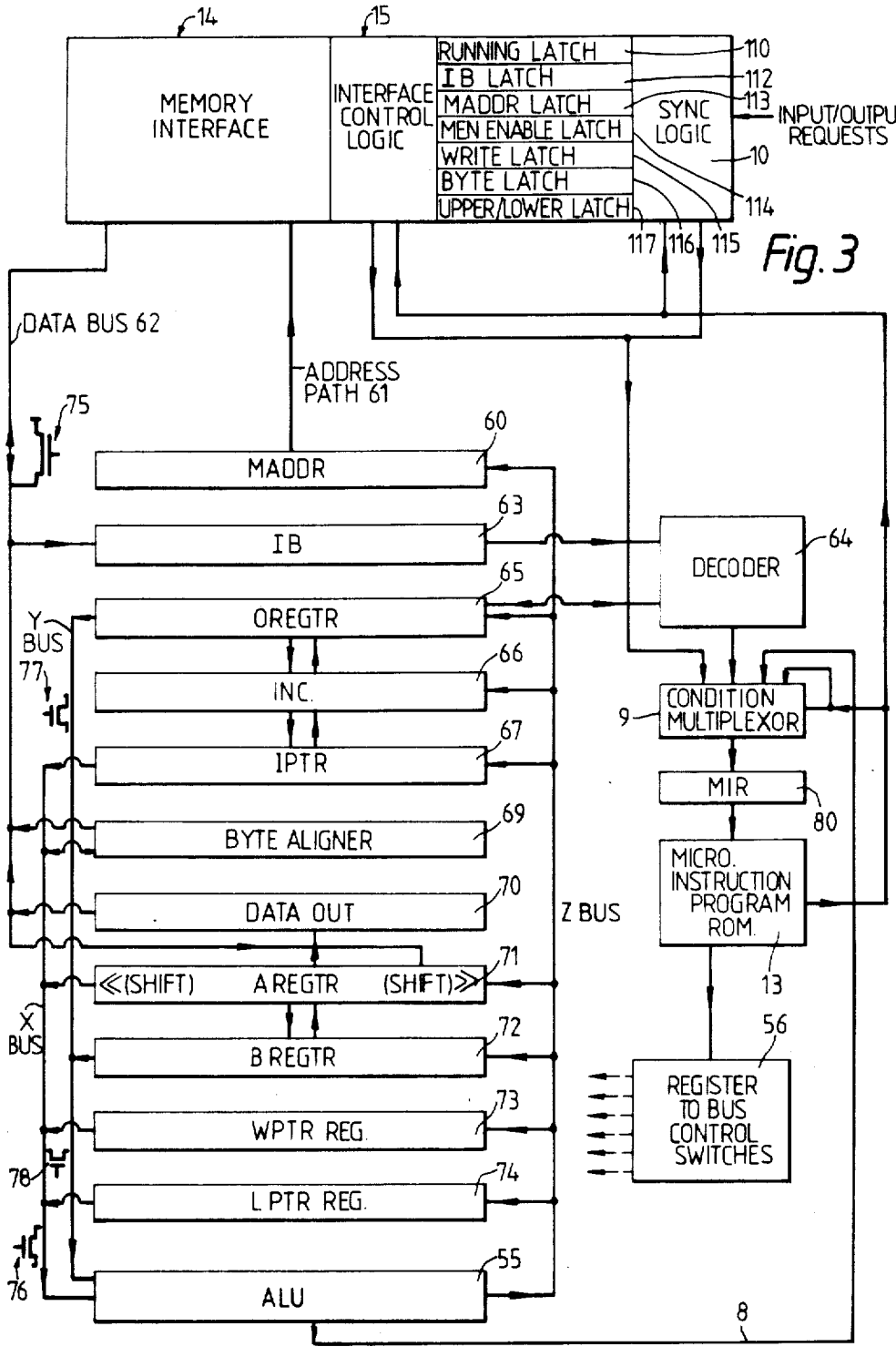

14
15
MEMORY INTERFACE
INTERFACE CONTROL LOGIC
RUNNING LATCH
IB LATCH
MADDR LATCH
MEN ENABLE LATCH
WRITE LATCH
BYTE LATCH
UPPER/LOWER LATCH
SYNC LOGIC
INPUT/OUTPUT REQUESTS
110
112
113
10
114
117
116
115
DATA BUS 62
ADDRESS PATH 61
75
MADDR
60
IB
63
OREGTR
65
Y BUS
77
INC.
66
IPTR
67
BYTE ALIGNER
69
DATA OUT
70
Z BUS
≪(SHIFT) AREGTR (SHIFT)≫
71
X BUS
B REGTR
72
WPTR REG.
73
78
L PTR REG.
74
76
ALU
55
8
DECODER
64
CONDITION MULTIPLEXOR
9
MIR
80
MICRO. INSTRUCTION PROGRAM ROM.
13
56
REGISTER TO BUS CONTROL SWITCHES

Fig. 3

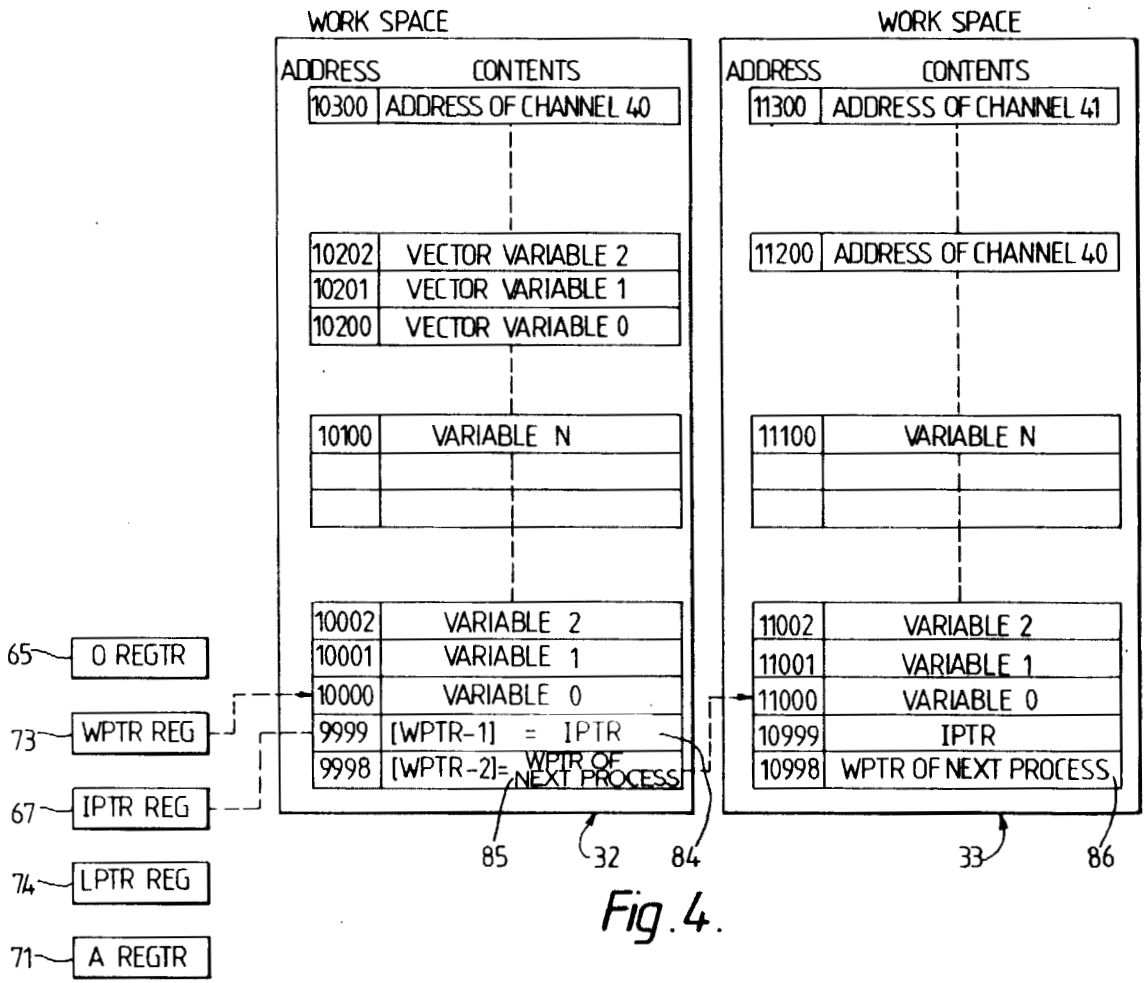

WORK SPACE
ADDRESS
CONTENTS
10300
ADDRESS OF CHANNEL 40
10202
VECTOR VARIABLE 2
10201
VECTOR VARIABLE 1
10200
VECTOR VARIABLE 0
10100
VARIABLE N
10002
VARIABLE 2
10001
VARIABLE 1
10000
VARIABLE 0
9999
[WPTR-1] = IPTR
9998
[WPTR-2]= WPTR OF NEXT PROCESS
WORK SPACE
ADDRESS
CONTENTS
11300
ADDRESS OF CHANNEL 41
11200
ADDRESS OF CHANNEL 40
11100
VARIABLE N
11002
VARIABLE 2
11001
VARIABLE 1
11000
VARIABLE 0
10999
IPTR
10998
WPTR OF NEXT PROCESS
65
0 REGTR
73
WPTR REG
67
IPTR REG
74
LPTR REG
71
A REGTR
85
32
84
33
86

Fig. 4.

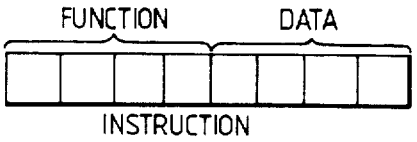


Fig. 5.

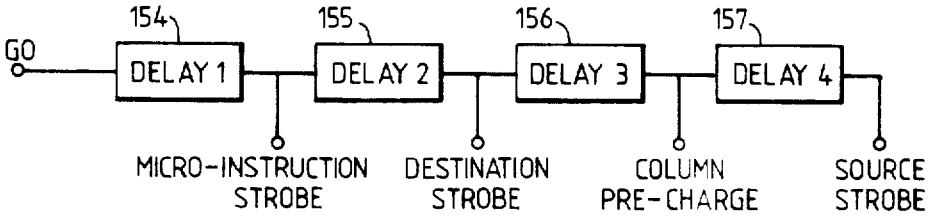


Fig. 7

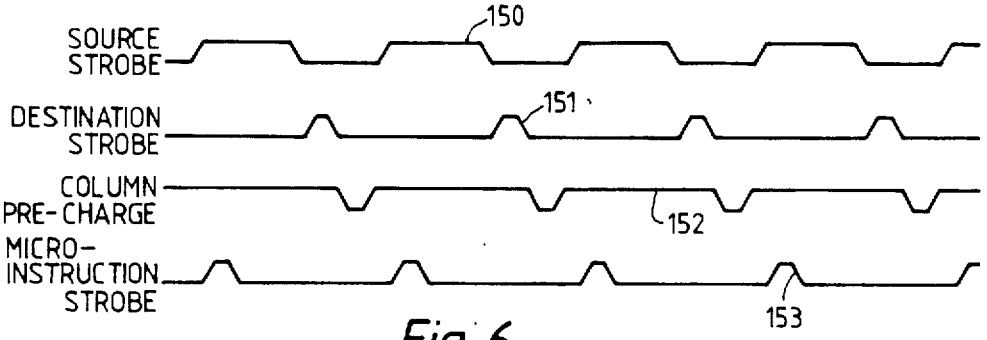
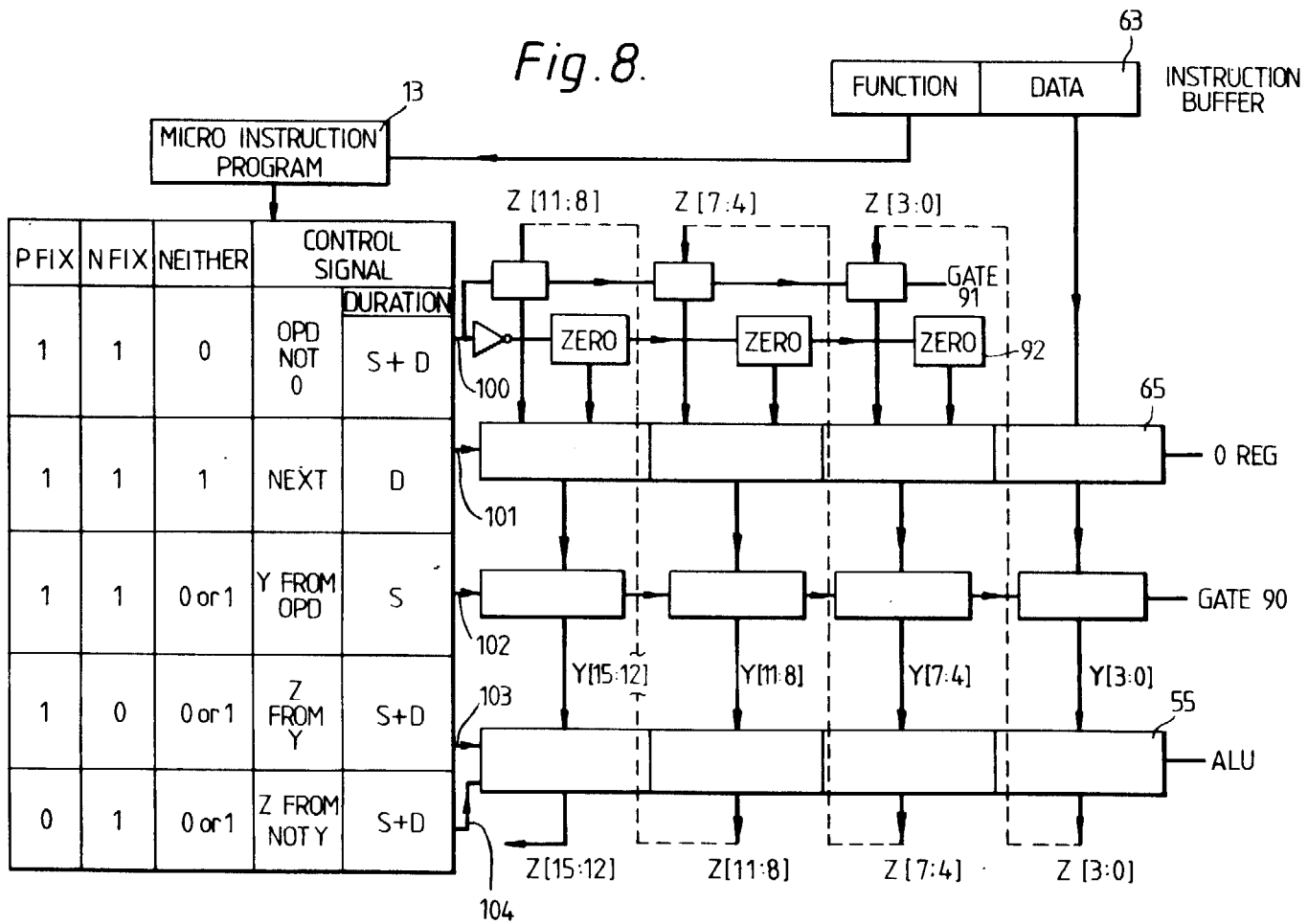


Fig. 6

Fig. 8.



PFIX	NFIX	NEITHER	CONTROL SIGNAL	DURATION
1	1	0	OPD NOT 0	S + D
1	1	1	NEXT	D
1	1	0 or 1	Y FROM OPD	S
1	0	0 or 1	Z FROM Y	S+D
0	1	0 or 1	Z FROM NOT Y	S+D

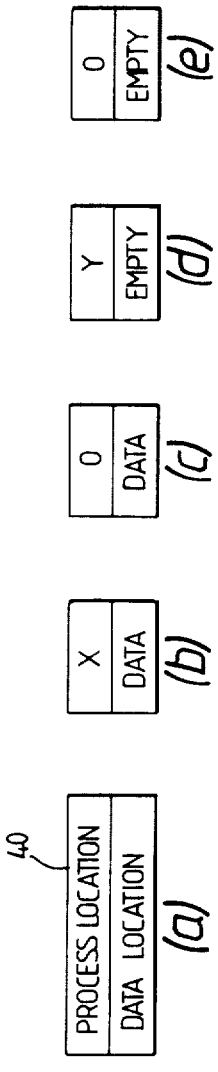


Fig. 9.

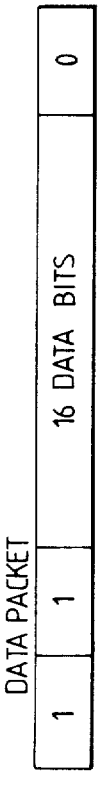


Fig. 13a.

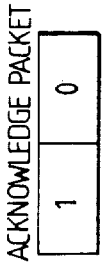


Fig. 13b.

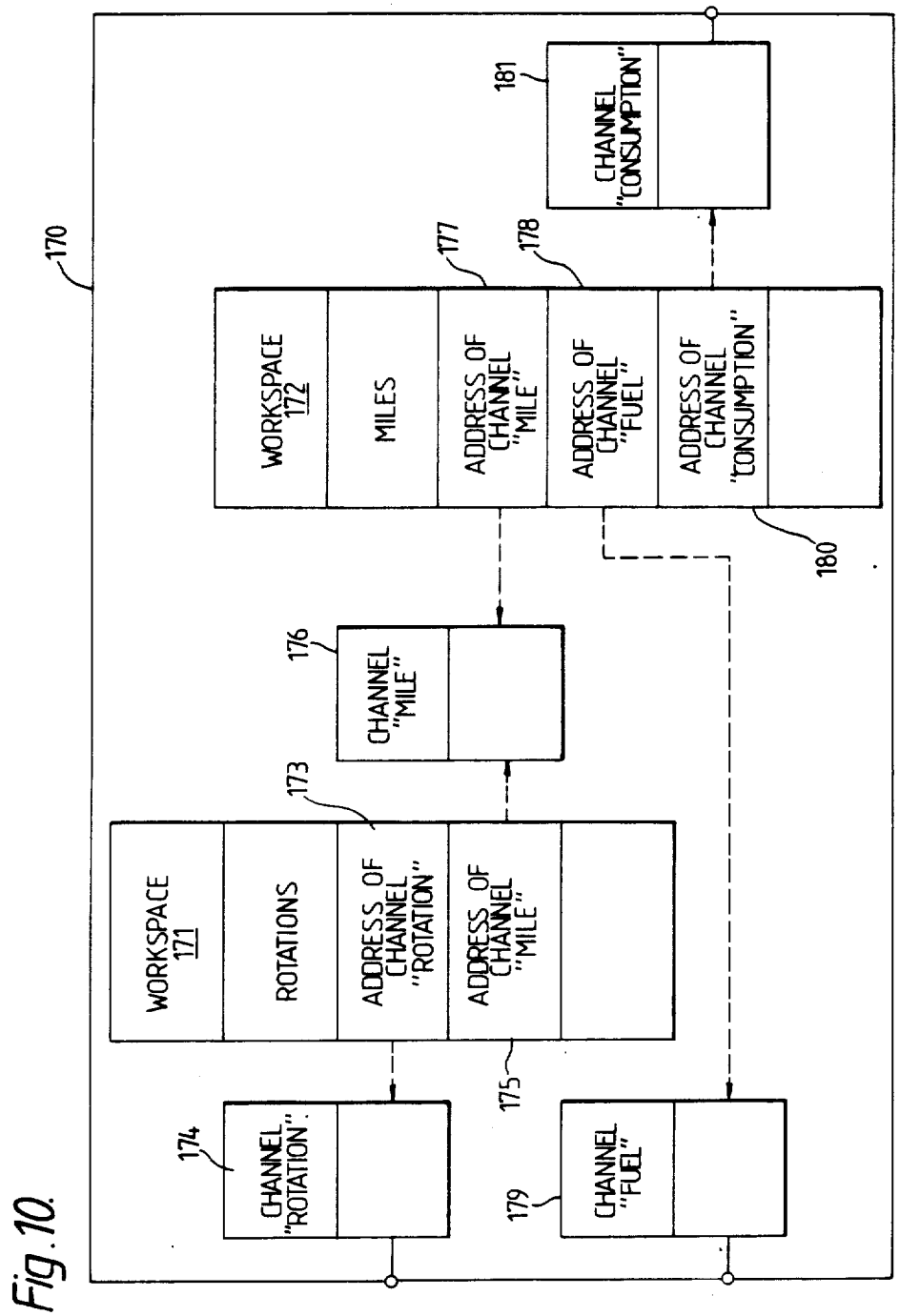

170
181
CHANNEL "CONSUMPTION"
177
178
WORKSPACE 172
MILES
ADDRESS OF CHANNEL "MILE"
ADDRESS OF CHANNEL "FUEL"
ADDRESS OF CHANNEL "CONSUMPTION"
180
176
CHANNEL "MILE"
173
WORKSPACE 171
ROTATIONS
ADDRESS OF CHANNEL "ROTATION"
ADDRESS OF CHANNEL "MILE"
175
174
CHANNEL "ROTATION"
179
CHANNEL "FUEL"

Fig. 10.

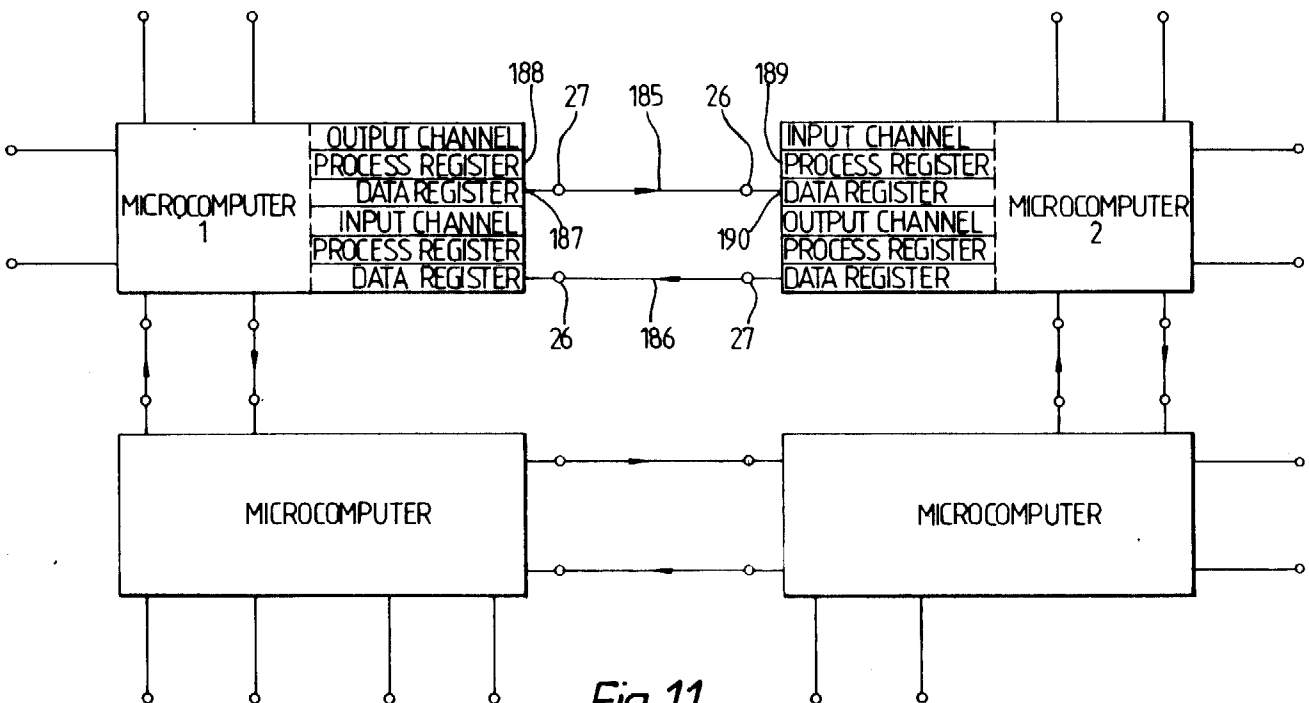

188
27
185
26
189
MICROCOMPUTER 1
OUTPUT CHANNEL
PROCESS REGISTER
DATA REGISTER
INPUT CHANNEL
PROCESS REGISTER
DATA REGISTER
187
190
INPUT CHANNEL
PROCESS REGISTER
DATA REGISTER
OUTPUT CHANNEL
PROCESS REGISTER
DATA REGISTER
MICROCOMPUTER 2
26
186
27
MICROCOMPUTER
MICROCOMPUTER

Fig. 11.

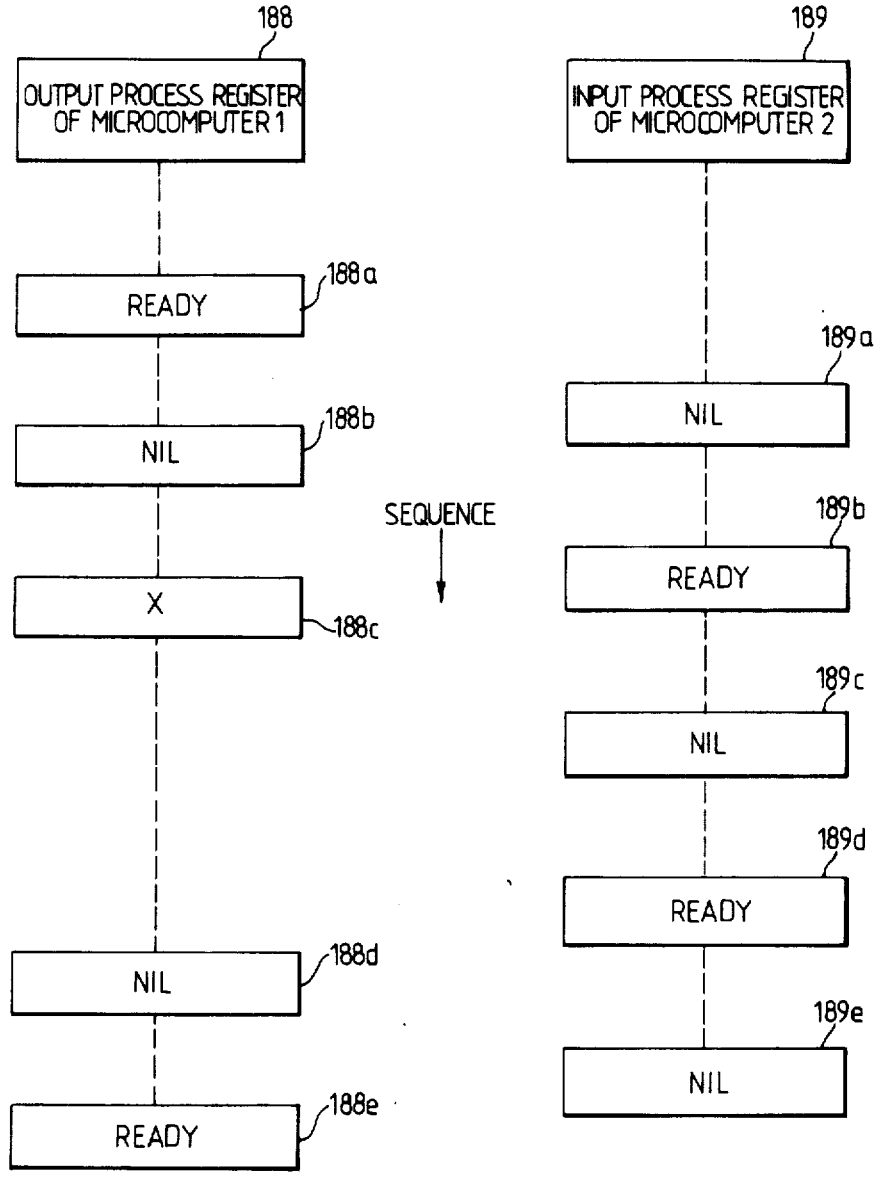

188
OUTPUT PROCESS REGISTER OF MICROCOMPUTER 1
READY
188a
NIL
188b
X
188c
NIL
188d
READY
188e
SEQUENCE
189
INPUT PROCESS REGISTER OF MICROCOMPUTER 2
NIL
189a
READY
189b
NIL
189c
READY
189d
NIL
189e

Fig. 12.

Fig. 14.

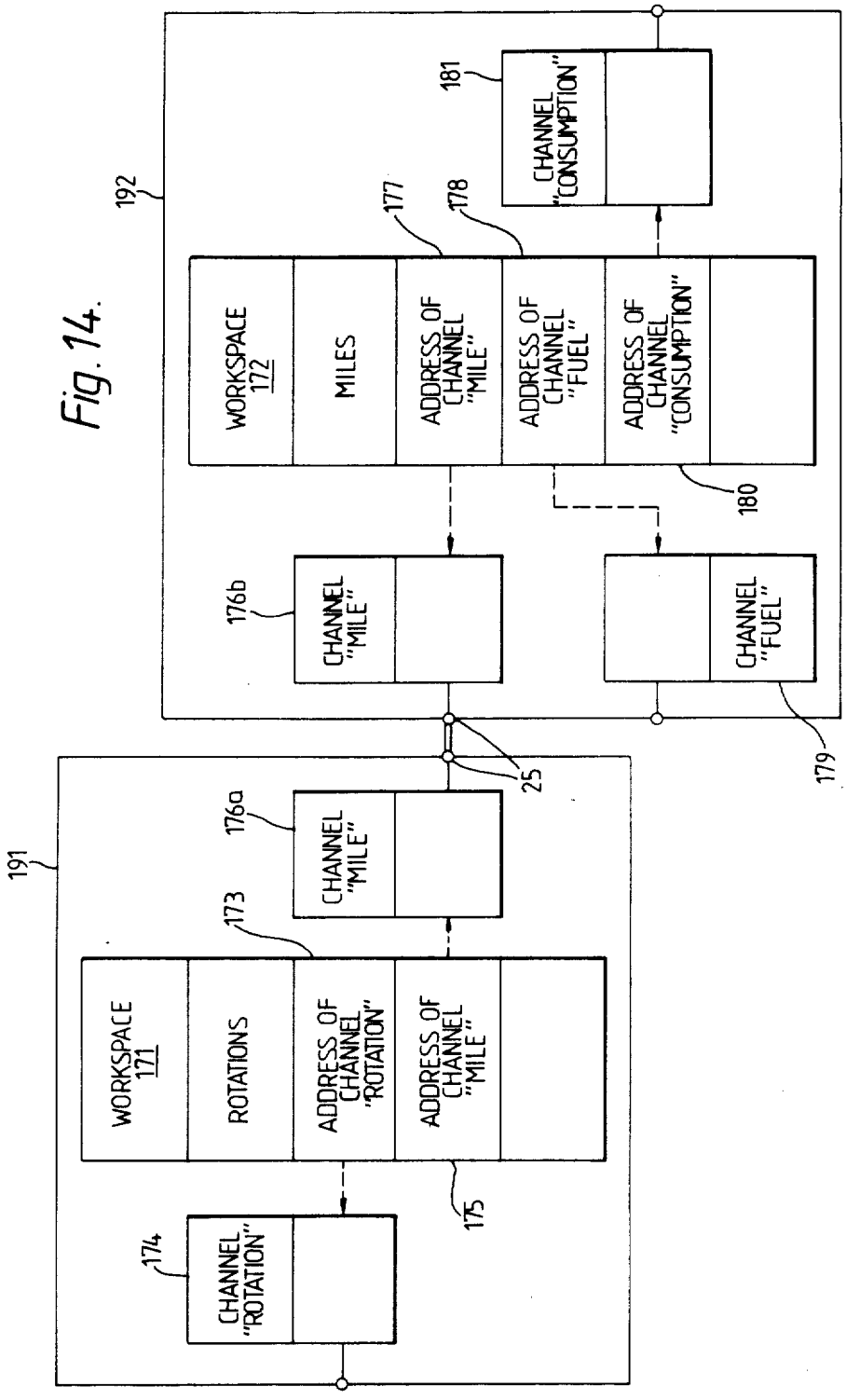

192
181
CHANNEL "CONSUMPTION"
177
178
WORKSPACE 172
MILES
ADDRESS OF CHANNEL "MILE"
ADDRESS OF CHANNEL "FUEL"
ADDRESS OF CHANNEL "CONSUMPTION"
180
176b
CHANNEL "MILE"
CHANNEL "FUEL"
179
25
191
176a
CHANNEL "MILE"
173
WORKSPACE 171
ROTATIONS
ADDRESS OF CHANNEL "ROTATION"
ADDRESS OF CHANNEL "MILE"
175
174
CHANNEL "ROTATION"

Fig. 15.

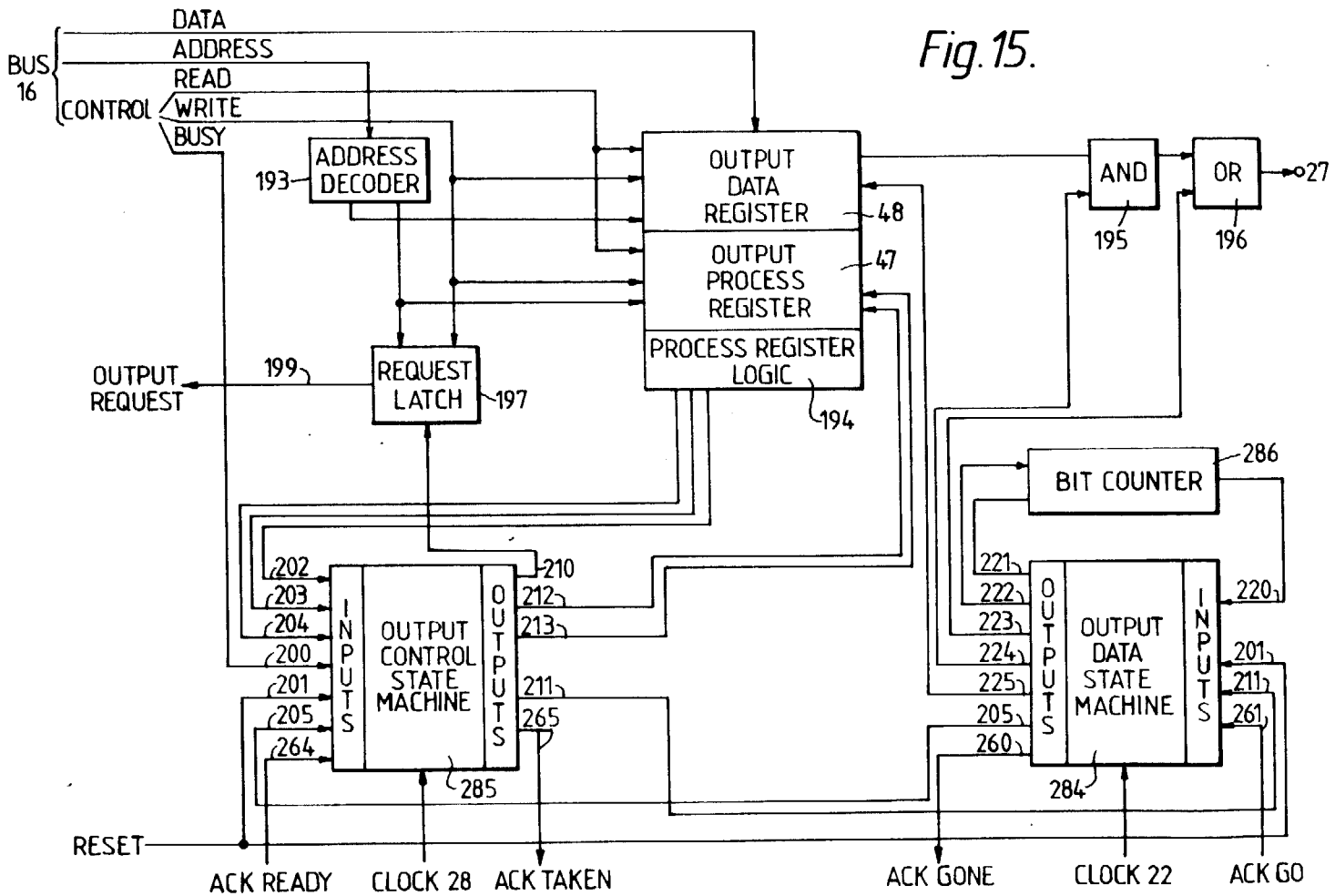
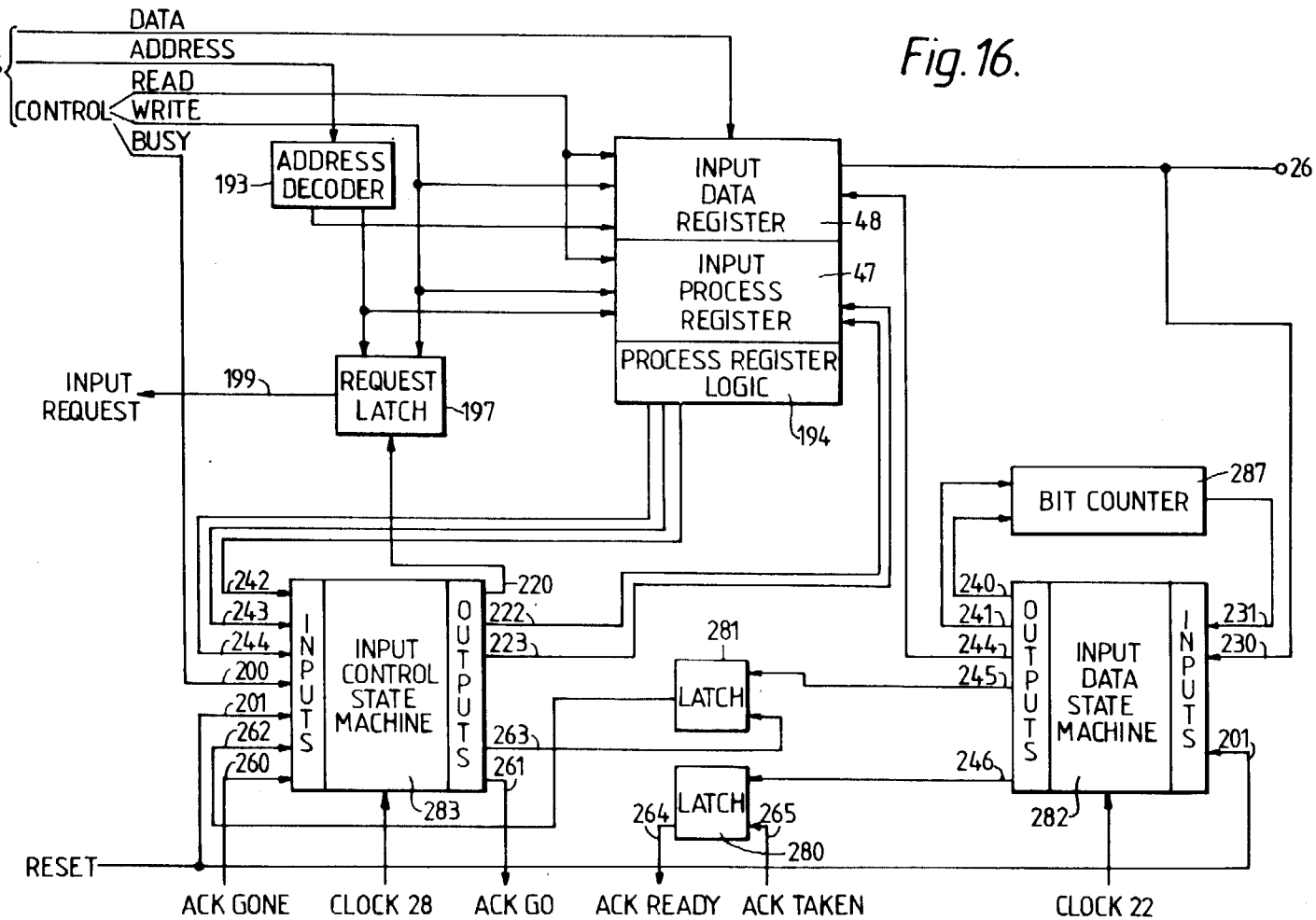

BUS 16
DATA
ADDRESS
READ
WRITE
BUSY
CONTROL
ADDRESS DECODER
193
OUTPUT DATA REGISTER
48
OUTPUT PROCESS REGISTER
47
PROCESS REGISTER LOGIC
194
AND
195
OR
196
27
OUTPUT REQUEST
199
REQUEST LATCH
197
BIT COUNTER
286
202
203
204
200
201
205
264
INPUTS
OUTPUT CONTROL STATE MACHINE
OUTPUTS
210
212
213
211
265
285
221
222
223
224
225
205
260
OUTPUTS
OUTPUT DATA STATE MACHINE
INPUTS
220
201
211
261
284
RESET
ACK READY
CLOCK 28
ACK TAKEN
ACK GONE
CLOCK 22
ACK GO

Fig. 16.


DATA
ADDRESS
READ
WRITE
BUSY
BUS 16
CONTROL
ADDRESS DECODER
193
INPUT DATA REGISTER
48
INPUT PROCESS REGISTER
47
PROCESS REGISTER LOGIC
194
26
INPUT REQUEST
199
REQUEST LATCH
197
BIT COUNTER
287
242
243
244
200
201
262
260
INPUTS
INPUT CONTROL STATE MACHINE
OUTPUTS
220
222
223
263
261
283
281
LATCH
LATCH
264
265
280
240
241
244
245
246
OUTPUTS
INPUT DATA STATE MACHINE
INPUTS
231
230
201
282
RESET
ACK GONE
CLOCK 28
ACK GO
ACK READY
ACK TAKEN
CLOCK 22

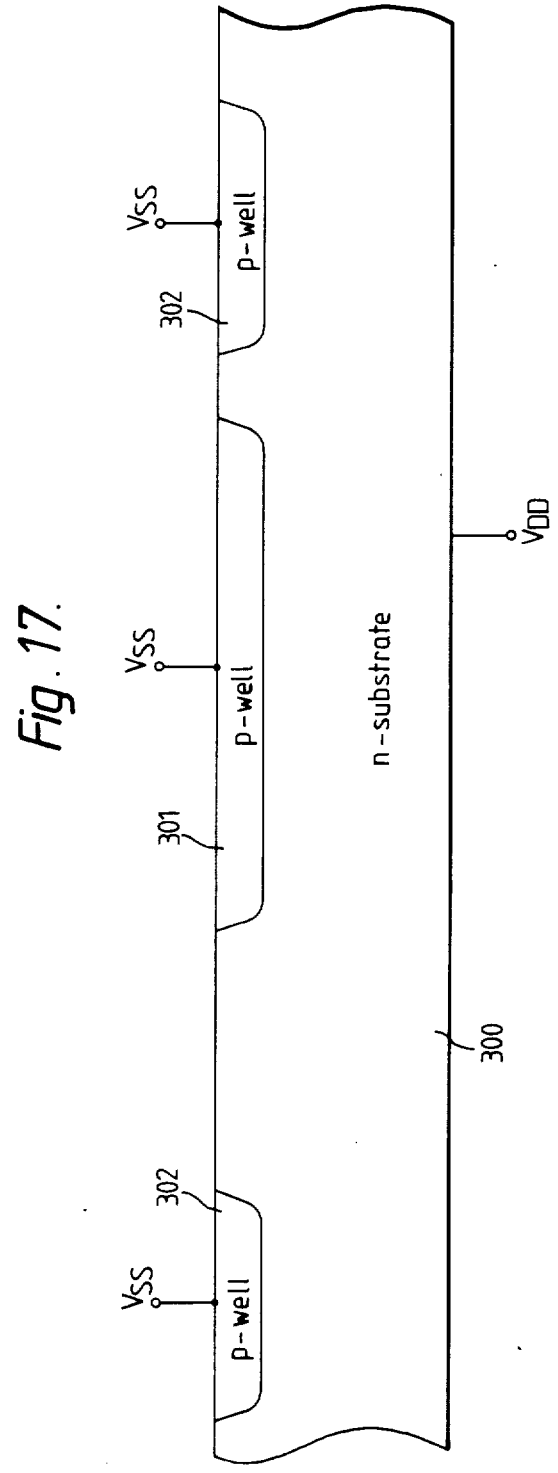


Fig. 17.

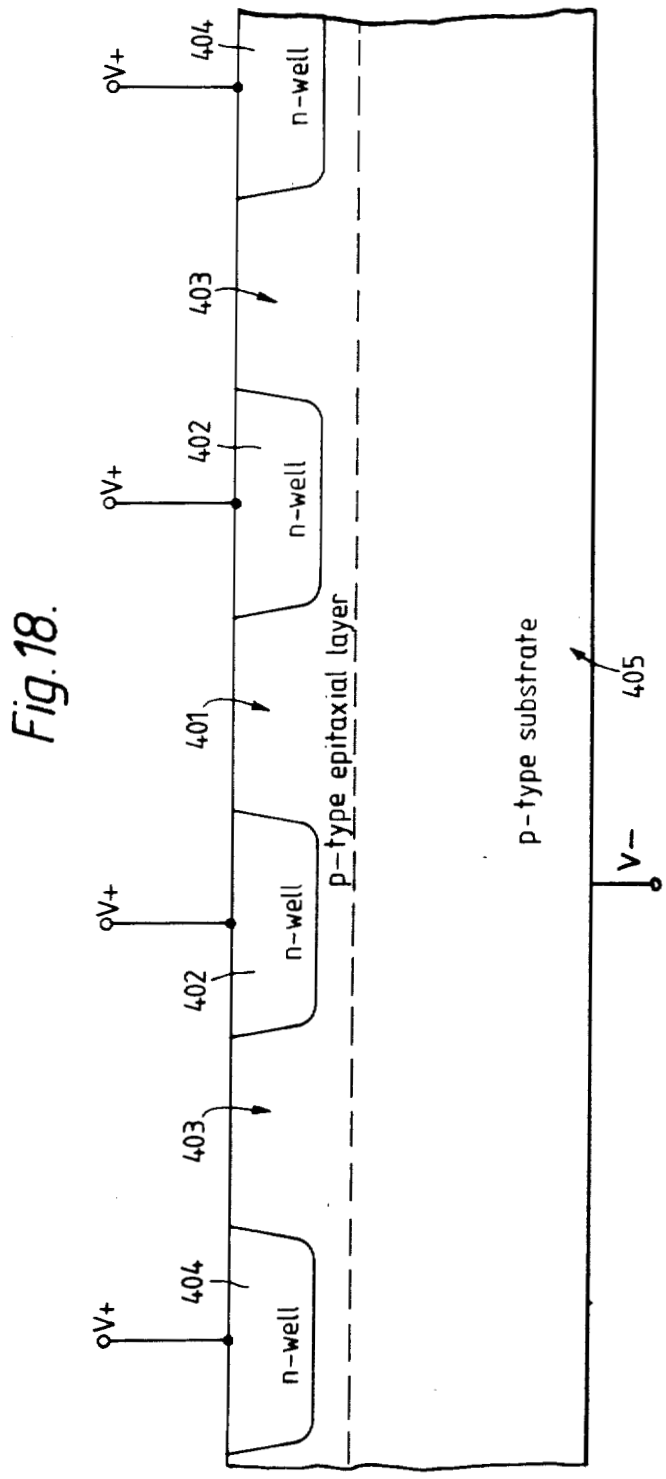

V+
404
n-well
403
V+
402
n-well
401
p-type epitaxial layer
p-type substrate
405
V−
V+
n-well
402
403
V+
404
n-well

Fig. 18.

HIGH DENSITY ROM IN SEPARATE ISOLATION WELL ON SINGLE WITH CHIP

The invention relates to microcomputers. This application is one of five applications assigned to Inmos Limited, each filed on Nov. 16, 1983 having Ser. Nos. 552,601; 552,602; 553,027; 553,028; and 553,029. These five applications have identical descriptions from pages 6 through 85.

BACKGROUND OF THE INVENTION

Microcomputers generally comprises a processor and memory, the processor being arranged to operate in accordance with the sequence of instructions derived from a stored program. In operation the processor will normally need to make data transfers between registers and between registers and memory. It may also wish to transmit messages to or from processes carried out on other microcomputers. The speed of access to memories located external to the processor chip is much slower. Furthermore, microcomputers have commonly arranged for external communications to take place through a shared bus which can act as a bottleneck causing slower operation of the microcomputer.

Although the development of integrated circuit technology has led to more components being provided on a microcomputer chip, difficulties arise in providing a useful amount of memory on the same chip as the processor.

The size available on a single silicon chip is limited and memory arrays may comprise the largest and densest component required for a microcomputer. Consequently the incorporation of any memory on the same chip as a pdrocessor may occupy excessive area and may lead to too low a yield of satisfactory components from a silicon chip due to the increased risk of defective components arising from the incorporation of memory on the chip.

Furthermore, difficulties arise in trying to avoid unacceptable noise interfering with the operation of a memory array when an asynchronous circuit such as a processor is incorporated on the same chip.

OBJECTS OF THE PRESENT INVENTION

It is an object of the present invention to provide an improved microcomputer with satisfactory combination of memory and processor on a single chip.

It is a further object of the invention to provide improved speed of operation of a microcomputer by reducing delay in transfers between processor and memory.

It is a further object of the present invention to provide an improved microcomputer in which a program for operating the microcomputer can be stored in a memory on the same chip as the processor.

It is a further object of the present invention to provide memory and processor on a chip with minimum noise interference.

It is a further object of the present invention to provide an improved microcomputer which incorporates memory and processor on the same chip with an acceptable yield in manufacturing processes.

SUMMARY OF THE PRESENT INVENTION

The present invention provides a microcomputer comprising a single chip having a processor and memory formed on the same chip, the memory comprising

an array of memory cells providing at least one K bytes (8K bits) of programmable RAM.

The present invention also provides a microcomputer comprising a single integrated circuit device providing a processor and memory, said processor being arranged to execute a number of operations on data in response to a program consisting of a plurality of instructions for sequential execution by the processor, each instruction including a set of function bits which designate a required function to be executed by the processor, wherein:

(a) said processor includes:

(i) plurality of registers and data transfer means for use in data transfers to and from said registers,

(ii) means for receiving each instruction and loading into one of the processor registers a value associated with the instruction, and

(iii) control means for controlling said data transfer means and registers responsive to said function bits to cause the processor to operate in accordance with said function bits and

(b) said memory comprises an array of memory cells providing at least one K bytes of RAM for storing a program to be executed by the processor.

Preferably the memory provides at least four K bytes of RAM.

Preferably said memory comprises a plurality of RAM cells formed with high impedance resistive loads and transistors.

In one embodiment said high impedance resistive loads are formed in a film of polycrystalline silicon.

Preferably the microcomputer has a substrate of semiconductor material and said memory is located in an isolation well formed of semiconductor material of different type from the substrate to reduce noise interference between the memory and processor.

Preferably the microcomputer comprises a CMOS structure having an n-channel substrate with one or more isolation wells of p-type semiconductor.

Preferably a plurality of isolation wells are provided in the substrate, the memory array being located in one isolation well and transistors of other circuitry used in the microcomputer being located in other isolation wells.

In order to achieve a satisfactory yield in manufacture, said memory array preferably comprises a main memory array and a redundant memory array, together with means for enabling use of redundant memory if defective memory elements occur in the main memory array.

Preferably said redundant memory array incorporates redundant rows and columns of memory elements interconnectable with the main memory array through fuse elements.

Preferably a single silicon chip on which is formed the processor, programmable RAM together with communication channels permitting message transmission to or from a process executed by the processor. Preferably said control means for the processor is arranged to respond to functions selected from a function set which include data transfer between registers, memory and registers and which enable synchronisation of message transfer through said communication channels.

Preferably said processor is arranged to execute a sequence of instructions each one byte long and each having the same format of bit positions, thereby reducing the chip area required by the processor registers,

said registers each having a bit length which is an integral number of bytes.

The invention also provides a network of microcomputers as aforesaid, the microcomputers being interconnected to execute a plurality of concurrent processes.

It will be understood that the term microcomputer does not impose any lower limit on how small the computer may be.

An example of a microcomputer in accordance with the present invention will now be described by way of example and with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram showing the main features of the microcomputer,

FIG. **2** is as more detailed block diagram of some of the components shown in FIG. **1** and in particular illustrates more fully the memory and serial links for external communication,

FIG. **3** shows further detail in block diagram form of part of the microcomputer and particularly illustrates the registers, data paths and arithmetic logic unit of the central processing unit as well as the interface between the central processing unit and other units of the microcomputer,

FIG. **4** illustrates the use of workspaces within the memory,

FIG. **5** illustrates schematically a form of instruction used in the microcomputer,

FIG. **6** shows in wave form the relative timing and duration of a plurality of timing control signals,

FIG. **7** illustrates the generation of timing control signals,

FIG. **8** illustrates the operation of the microcomputer of FIGS. **1** to **3** with variable length operands.

FIGS. **9***a* to **9***e* illustrate successive operations in one manner of communicating using a two word channel between two processes which are executed by the same microcomputer,

FIG. **10** illustrates the operation of two communicating processes on one microcomputer,

FIG. **11** shows a network of interconnected microcomputers, including detail of the serial link connection between two of them,

FIG. **12** illustrates a sequence of operations for effecting communication via serial links between two processes carried out on different microcomputers,

FIGS. **13***a* and **13***b* illustrate the format of data and acknowledge packets for transmission through serial links between two microcomputers,

FIG. **14** illustrates the operation of the same two communicating processes of FIG. **10** on two interconnected microcomputers,

FIG. **15** shows a logic diagram of one output serial link,

FIG. **16** shows a logic diagram of one input serial link,

FIG. **17** shows the chip formation which may be used for the microcomputer of FIG. **1**, and

FIG. **18** shows an alternative chip formation which may be used for the microcomputer of FIG. **1**.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The microcomputer described herein is an example of a Transputer (Trade Mark of Inmos International plc) microcomputer and comprises a single silicon chip having both a processor and memory as well as links to permit external communication. It is capable of carrying out a plurality of concurrent processes and effecting scheduling and communication between processes both on the same chip or separate chips. Each microcomputer has at least one K byte of memory in the form of programmable RAM on the same chip as the processor and the processor is capable of obeying programs in the chip's memory. The microcomputer has a plurality of communication links, herein called serial links, to enable it to be connected into a network of interconnected microcomputers so that any one microcomputer can be used as a building block for a network. The communication between any two microcomputers is effected by a serial link which provides one or more specific pin to pin connections each interconnecting two and only two microcomputers. Each link is not shared with other microcomputers or with any external memory. The microcomputer is provided with means for synchronisation in data transmission between microcomputers within the network so that communication through a link between two microcomputers can be initiated by either the receiving or transmitting microcomputer.

The microcomputer contains a program with a plurality of sequential instructions each consisting of two parts, one part representing the function of the instruction and the other part representing data which is loaded into an operand register. In this way the function part of each instruction is of the same bit length regardless of the word length of the processor and in this way uniformity of function format and function bit length is achieved regardless of the word length of the processor. A further important feature of the microcomputer is that its operation is effected by use of a function set which is simple and efficient. The function set consists of a minimum number of functions. The function set includes direct functions which cause the processor to carry out an operation on the contents of the operand register. In a preferred arrangement it also includes one indirect function and two prefixing functions. The use of the indirect function allows a large number of processor operations to be used without increasing the number and size of data registers to perform the operations. Furthermore the use of a prefixing function provides for variable length operands.

By use of a microcomputer in accordance with this example, any required network of microcomputers can be formed by interlinking a number of microcomputers and the resulting network operates in the same way as any single microcomputer.

GENERAL DESCRIPTION OF THE STRUCTURE

The main elements of the microcomputer are illustrated in FIG. **1** on a single silicon chip **11** using p-well complementary MOS technology, which will be described in more detail with reference to FIG. **17**. The components provided on the chip have been indicated in block form in FIG. **1** although it will be appreciated that the blocks are not intended to represent the relative size and positioning of the various components. On the chip there is provided a central processing unit (CPU) **12** which includes some read-only memory (ROM) **13**. The CPU **12** is coupled to a memory interface **14** controlled by interface control logic **15**. The CPU **12** incorporates an arithmetic logic unit (ALU), registers and data paths which will be described in more detail with reference to FIG. **3**. The CPU **12** and memory interface

14 are connected to a bus 16 which provides interconnection between the elements on the chip 11. A service system 17 is provided with a plurality of input pins 18 including a zero volt supply, a 5 volt supply, a reset pin which may be activated to reset the microcomputer to a defined state, and a clock pin 28. The microcomputer is provided with a substantial amount of memory on the chip 11 and this is represented by a random-access memory RAM 19 and the ROM 20. The amount of memory on the chip should not be less than 1K byte so as to provide sufficient memory capacity to allow the processor 12 to be operated without external memory. Preferably the memory on the chip is at least 4K bytes. The division between RAM and ROM on the chip may be selected to suit the particular requirements for the microcomputer. The memory also includes redundancy 21 (this may be as described in U.S. Pat. No. 4,346,459 entitled "Redundancy Scheme For An MOS Memory;" U.S. Pat. No. 4,389,715 entitled "Redundancy Scheme For A Dynamic RAM;" or U.K. Patent application 8231055, all owned by Inmos Corporation. This region 21 of memory has rows and columns selectively connectable by fuses as shown to replace defective regions of the memory 19 or 20 and thereby increase the production yield of chips which are satisfactory for use. The operation of the microcomputer includes timing control responsive to clock pulses from the pin 28. An external memory interface 23 is provided and connected to a plurality of pins 24 for connection to an optional external memory (not shown). In order to allow the microcomputer to be linked to other similar microcomputers to form a network, a plurality of serial links 25 are provided and in this example four are shown. Each serial link 25 has an input pin 26 and an output pin 27 each of which can be used to form a single pin to pin connection to corresponding output and input pins respectively of a further microcomputer. Each serial link is connected to a synchronisation logic unit 10 comprising process scheduling logic which will be described in more detail below. Although the drawings show four serial links 25, three links, or even two links, may be used to form a single network but preferably at least six, and for example seven, such links are provided so that they may be fully interconnected in any desired array.

GENERAL DESCRIPTION OF USE OF CHIP MEMORY AND COMMUNICATION CHANNELS AND LINKS

FIG. 2 shows some of the elements of the microcomputer in more detail and in particular it illustrates the use of the memory on the chip. The microcomputer may be used to carry out a plurality of concurrent processes on the same chip and in FIG. 2 the operations of three concurrent processes have been shown. The memory is used to store the program 30 which may be stored in either ROM 20 or RAM 19. In this particular example the microcomputer is a 16 bit word device although it will be understood that other word lengths may be used. The program 30 consists of a sequence of instructions which in this example are each of 8 bit length and this instruction length may remain the same even if the processor is of word length other than 16 bits. Each instruction is of the format shown in FIG. 5 where the most significant 4 bits represent the function of the instruction and the least significant 4 bits represent data. The program 30 incorporates no data other than that held in the designated part of each instruction. The manner in which the processor responds to each function and the way in which the data is handled depend on the particular function selected from a set of functions which will be described below, but the format of the function and data parts of each instruction is always the same. The memory also stores data 31 which may be stored in either the ROM 20 or RAM 19.

The microcomputer carries out a number of processes together, sharing its time between them. Processes which are carried out together are called concurrent processes. At any time, only one of the processes is actually being executed by the microcomputer and this process is called the current process. Each concurrent process to be effected by the microcomputer uses a region of memory called a workspace for holding the local variables and temporary values manipulated by the process. The address of the first local variable of each workspace is indicated by a workspace pointer (WPTR). Similarly for each concurrent process, an instruction pointer (IPTR) is used to indicate the next instruction to be executed from the sequence of instructions in the program relating to that particular process. In FIG. 2, which shows three concurrent processes, the workspace for process 1 is indicated by the numeral 32 and the corresponding workspaces for processes 2 and 3 have been marked 33 and 34. Each workspace consists of a plurality of addressable word locations and one word location 35 of each workspace is used to store the workspace pointer (WPTR) of the next process to be executed on a list of processes waiting to be executed. Thus, a linked list is formed in memory containing pointers to a sequence of workspaces for processes to be executed. If the processor is working on process 1 (see FIG. 2) and reaches a point where it is instructed that for the time being it is to stop executing that process, the CPU 12 will begin work on the next process, e.g. process 2. It will be directed to that next process by reading the workspace pointer in memory at location 35. In the preferred embodiment there is a known relationship between workspace pointer for any process and the address of the workspace pointer of the next process on the linked list, so that the next part of the linked list will be easily available from the current process workspace. For each process workspace, a further word location 36 stores the instruction pointer (IPTR) for that process. It will be appreciated that although workspaces for only three processes are shown in FIG. 2, the number may be varied depending on the number of concurrent processes to be carried out.

In order to allow communication between different processes carried out by the same microcomputer, a plurality of communication channels indicated by the numerals 40, 41, 42 and 43 are provided in the RAM section 19 of the memory. In this example each communication channel consists of two word locations in memory, one for use in identifying the process wishing to use the channel and the second for holding the data to be communicated through the channel. The operation of these channels will be described more fully with reference to FIGS. 9a–9e. FIG. 2 also shows in more detail the formation of one serial link 25. It is to be understood that each of the serial links is similarly formed. As indicated, the link 25 incorporates two channels 45 and 46 each forming a uni-directional communication channel. In this way the channel 45 is used as an input channel and the channel 46 as an output channel. Each channel consists of two registers each addressable in a manner similar to the two word locations of each of the chan-

nels **40** to **43**. The two registers consist of a process register **47** used to indicate the process involved in the communication and a data register **48** for holding the data to be transmitted. The data register **48** in the input channel is connected to pin **26** and the data register **48** in the output channel is connected to pin **27**. The operation of the two registers **47** and **48** is controlled by control logic **50** coupled to the synchronisation unit **10**. The operation of the serial links, control logic **50** and unit **10** will be described in more detail with reference to FIGS. **12** to **16**.

The RAM section **19** of the memory is used to provide the workspaces **32** to **34** as well as the communication channels **40** to **43** and it may also be used for holding the program and data if required. The ROM **20** may be used for a variety of purposes such as for example holding an interpreter for a high level programming language or for storing "look-up" tables for standard operations. It may also be used to hold control programs for peripheral devices where the microcomputer is intended for a specific purpose.

CPU DATA PATHS AND REGISTERS

The central processing unit **12** and its operation will be more fully understood with reference to FIG. **3**.

The CPU **12** includes an arithmetic logic unit (ALU) **55** and a plurality of data registers connected to three data buses, X bus, Y bus and Z bus. The operation of the registers and their interconnections with the buses is controlled by a plurality of switches diagrammatically represented by the reference numeral **56** and controlled by signals derived from a micro-instruction program contained in the ROM **13**. It will be understood that these are switch means integrally formed in the chip construction. Communication between the CPU and the memory (**19**, **20**, **21**) is effected via a unidirectional address path **61** leading to the memory interface **14** and a bidirectional data bus **62** also connected to the interface **14**. The registers, buses **61** and **62**, and the X, Y and Z buses are connected as shown in FIG. **3**. The registers are as follows:

Abbreviation	Register
MADDR	Memory address register 60 containing the address of the memory location required.
IB	Instruction buffer 63 for receiving sequentially from memory instructions of the form shown in FIG. 5.
OREGTR	An operand register 65 for receiving the data derived from an instruction in the instruction buffer 63.
IPTR REG	A register 67 which holds the instruction pointer (IPTR) of the current process.
DATA OUT	A register 70 for supplying data to the memory on the data bus 62.
AREGTR	A first (A) register 71 for holding an operand for the ALU 55.
BREGTR	A second (B) register 72 arranged as a stack with the AREG for holding operands for the ALU 55.
WPTR REG	A register 73 for holding the workspace pointer (WPTR) of the current process.
LPTR REG	A register 74 for holding a pointer to the workspace of the last process on the list of processes waiting to be executed.

As shown in FIG. **3**, an incrementer **66** and a byte aligner **69** are also provided.

The data bus **62** is provided with a switch **75** operable to precharge the data bus line **62**. The X and Y buses are respectively provided with similar switches **76** and **77** operable to precharge the X and Y buses. A further switch **78** is provided between the X and Y buses and is operable to cause signals on the two buses to merge.

The arithmetic logic unit **55** receives inputs from both the X and Y buses and is arranged to output to the Z bus. It provides a further output **8** to the micro-instruction program ROM **13**, through a condition multiplexor **9**, so as to control the operation of the data path in dependence on the output of the ALU **55**.

The instruction buffer **63** is arranged to receive from the memory (**19**, **20**, **21**) via interface **14** and bus **62** a sequence of 8 bit words, herein called instructions, each of which has the format shown in FIG. **5** and consists of two parts. One part represents a "function" selected from the function set described below and the other part represents data. The instruction buffer **63** provides an output to a decoder **64** which separates the instruction into the function and data halves. The data half is loaded into the operand register **65** and the function half is decoded to provide an address to a micro-instruction register (MIR) **80**. The identical procedure is followed for all instructions, regardless of function selected. Each instruction received by the instruction buffer **63** loads into the MIR **80** an address which causes the micro-instruction program in the ROM **13** to execute one or more micro-instructions controlling the switches **56** and interface control logic **15** so that at the end of each sequence of micro-instructions, an operation has been effected by the registers, control logic **15**, and data paths of FIG. **3** corresponding to the selected function in the instruction. The operation of the micro-instruction program will be described more fully below.

All the registers shown in FIG. **3** apart from the instruction buffer **63** and the micro-instruction register **80** are 16 bit registers. It will be appreciated that in this example in which the processor is a 16 bit word processor, each 16 bit word location in the program contains two instructions, as each instruction is only 8 bits long. It is therefore necessary for the instruction pointer, which is held in the register **67** to be capable of pointing to a specific 8 bit byte in order to identify a single instruction from a program word location which incorporates two instructions. For this reason the program **30** (FIG. **2**), in this example, is written into the bottom half only of the memory **19**. In this example the memory has 64K words and consequently the program **30** is written into locations 0 to 32767 as the addresses of these locations can be represented by 15 bits only. This leaves an additional bit in the instruction pointer which can be used to identify which of the two bytes at each word address is necessary in order to identify a specific instruction. The micro-instruction ROM **13** contains 122 words, each of 68 bits. Each row of the ROM **13** contains 68 bits so that the ROM is arranged to provide 68 output signals at any time. The operation of the micro-instruction program will be described more fully below.

As can be seen from FIG. **3**, the interface logic controller **15** is provided with a plurality of single bit state latches which are used to record the state of the memory interface. A latch **110**, called a running latch, defines the source of instructions to be executed. If the latch **110** has state **1** the source of instructions is memory (this may be an external memory via the external memory interface **23** if desired). If the latch has state **0**, the source of instructions is one of the serial links **25** to allow instructions to be received from an external source. It may be necessary to go repeatedly to the same serial link **25** for two or more successive instructions whereas when the instructions are derived from mem-

ory, the instruction pointer IPTR is advanced for each instruction. An IB latch **112** records the state of the IB register **63**. An MADDR latch **113** records the state of the MADDR register. A MEM ENABLE latch **114** records the state of the memory interface and has state **1** whenever the memory interface **14** is occupied. A WRITE latch **115** records that a write request has been made to the memory. The BYTE latch **116** records that a byte request has been made to the memory. An UPPER/LOWER latch **117** holds the least significant bit of byte addresses and is loaded from the least significant bit of the A register **71** when the content of the A register is shifted one place to the right.

FUNCTION SET

The function elements of the instructions which are received by the instruction buffer **63** are determined by the function set for the microcomputer. The function set is the list of available functions which can be selected when writing a program and to which the microcomputer is capable of responding.

There are three types of function in the function set.

Direct functions which use the contents of the operand register **65** as data (the contents of other registers may also be used as data).

Indirect functions which use the contents of the operand register **65** to select one of a variety of "operations" using data in registers other than the operand register **65**. The selectable "operations" are listed below the function set.

Prefixing functions which accumulate operands into the operand register **65**.

The function set is as follows:

Code No	Abbreviation	Name
		FUNCTIONS
		Direct Functions
0	ldw	load from workspace
1	stw	store to workspace
2	ldpw	load pointer into workspace
3	ldwi	load from workspace and increment
4	ldv	load from vector
5	stv	store to vector
6	ldl	load literal
7	adl	add literal
8	j	jump
9	jnz	jump non zero
10	ldpc	load pointer into code
11	call	call procedure
		INDIRECT FUNCTIONS
13	opr	operate
		PREFIXING FUNCTIONS
14	pfix	prefix
15	nfix	negative prefix

The operations which may be effected by use of indirect functions are as follows:

	OPERATIONS	
Code No	Abbreviation	Name
0	rev	reverse
1	eqz	equal to zero
2	gt	greater
3	and	and
4	or	or
5	xor	exclusive or
6	add	add
7	sub	subtract
8	run	run process
9	pse	pause
10	join	join
11	sync	sychronise
12	ret	return
13	rot	rotate bytes
14	sr	shift right
15	sl	shift left

Prior to describing these functions and operations, the notation which is used herein will be set forth. The Transputer microcomputer is used preferably with OCCAM (Trade Mark of Inmos International plc) language, which is set forth more particularly in the booklet entitled *Programming Manual-OCCAM* published and distributed by Inmos Limited in 1983 in the United Kingdom, a copy of which is attached to this specification as Appendix 1 as well as Taylor and Wilson, "Process—Oriented Language Meets Demands of Distributed Processing", *Electronics* (Nov. 30, 1982), both of which are hereby incorporated herein by reference. OCCAM language is particularly well suited to concurrent processing. Because the preferred embodiment is particularly suitable for concurrent processing, the use of OCCAM language with the present example is quite appropriate.

Other languages can be used with an appropriate compiler. In actual application, the programmer will write a program using OCCAM language and a compiler will convert this to particular instructions in customary fashion. Nevertheless, the functions and operations in the instructions are susceptible of description using OCCAM language to show what happens within the preferred embodiment of the microcomputer described herein. Thus, in describing these functions and operations, as well as examples of use, the following notation will be used:

NOTATION

1. PROCESS

A process starts, performs a number of actions, and then terminates. Each action may be an assignment, an input or an output. An assignment changes the value of a variable, an input receives a value from a channel, and an output sends a value to a channel.

At any time between its start and termination, a process may be ready to communicate on one or more of its channels. Each channel provides a one way connection between two concurrent processes; one of the processes may only output to the channel, and the other may only input from it.

An assignment is indicated by the symbol ":=". An assignment

```
v:=e
```

sets the value of the variable v to the value of the expression e and then terminates. For example, x:=0 sets x to zero, and x:=x+1 increases the value of x by 1.

An input is indicated by the symbol "?". An input

```
c?x
```

inputs a value from the channel "c", assigns it to the variable x and then terminates. An input

```
c?ANY
```

inputs a value from the channel "c", and discards the value.

An output is indicated by the symbol "!". An output

```
c!e
```

outputs the value of the expression e to the channel "c" and then terminates. An output

```
c!ANY
```

outputs an arbitrary value to the channel "c".

The process SKIP terminates with no effect.

2. CONSTRUCT

A number of processes may be combined to form a sequential, parallel, conditional or alternative construct. A construct is itself a process, and may be used as a component of another construct. Each component process of a construct is written two spaces further from the left hand margin, to indicate that it is part of the construct.

A sequential construct is represented by

```
SEQ
  P1
  P2
  P3
  ...
```

The component processes P1, P2, P3 . . . are executed one after another. Each component process starts after the previous one terminates and the construct terminates after the last component process terminates. For example

```
SEQ
  in ? x
  x := x + 1
  out ! x
```

inputs a value, adds one to it, and then outputs the result.

A parallel construct is represented by

```
PAR
  P1
  P2
  P3
  ...
```

The component processes P1, P2, P3 . . . are executed together, and are called concurrent processes. The construct terminates after all of the component processes have terminated. For example,

```
PAR
  in ? x
  out ! y
```

allows an input to x and output from y to take place together.

Concurrent processes communicate using channels. When an input from a channel "c", and an output to the same channel "c" are executed together, communication takes place when both the input and the output are ready. The value is assigned from the outputting process to the inputting concurrent process, and execution of both concurrent processes then continues.

A conditional construct

```
IF
  condition 1
    P1
  condition 2
    P2
  condition 3
    P3
  ...
```

means that process P1 is executed if condition 1 is true, otherwise process P2 is executed if condition 2 is true, and so on. Only one of the processes is executed, and the construct then terminates. For example

```
IF
  x >= 0
    y := y+1
  x < 0
    SKIP
```

increases y only if the value of x is positive.

An alternative construct

```
ALT
  input 1
    P1
  input 2
    P2
  input 3
    P3
  ...
```

waits until one of input 1, input 2 . . . is ready. If input 1 first becomes ready, input 1 is performed, and then process P1 is executed. Similarly, if input 2 first becomes ready, input 2 is performed, and then process P2 is executed. Only one of the inputs is performed, and then the corresponding process is executed and the construct terminates. For example:

```
ALT
  count ? ANY
    counter := counter + 1
  total ? ANY
    SEQ
      out ! counter
      counter :=0
```

either inputs a signal from the channel "count", and increases the variable "counter" by 1, or alternatively inputs from the channel "total", and outputs the current value of the variable "counter", and resets it to zero.

3. REPETITION

```
WHILE condition
  P
```

repeatedly executes the process P until the value of the condition is false. For example

```
WHILE x > 5
  x := x - 5
```

decreases x by 5 until its value is less than 5.

4. VARIABLES

A variable is either a simple variation, corresponding to a single word in store, or is one of a numbered set of variables called a vector. For example, v [3]:=0 sets the value of variable number 3 in the vector v to 0, and v[0]+1 increases the value of variable number 0 by 1.

A variable is introduced by a declaration such as

```
VAR v :
P
```

which introduces v for use in the process P.

5. PROCEDURES

A procedure definition allows a process to be given a name. For example

```
PROC square ( n, sqr )
  sqr := n * n
```

defines the procedure "square".

The procedure name may be used as an abbreviation for the process. For example

```
square (x, sqrx)
```

means

```
sqrx:=x*x
```

6. EXPRESSIONS

An expression is constructed from operators, variables, numbers, the truth values TRUE and FALSE and the brackets (and).

TRUE is a value consisting entirely of 1 bits, and FALSE is a value consisting entirely of 0 bits.

The operators +, −, *, / represent addition subtraction, multiplication and division as usual.

For the operators =, <>, > and <=, the result is produced as shown below:

x=y true if x is equal to y

x<>y true if x is not equal to y

x>y true if x is greater than y

x<=y true if x is less than or equal to y

For the operators \/, /\ and ><, each bit of the result is produced from the corresponding bits of the operands according to the following table:

x	y	x \/ y	x /\ y	x >< y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

For the NOT operator, each bit of the result is produced from the corresponding bit of the operand, according to the following table:

x	NOT x
0	1
1	0

For the operators << and >>

x<<y is the value of x moved y bits to the left, vacated bit positions being filled with 0 bits

x>>y the value of x moved y bits to the right vacated bit positions being filed with 0 bits

The above general OCCAM language notation will now be applied to the microcomputer of the example.

The register variables are defined as follows:

IPTR represents the contents of the instruction pointer register **67**

WPTR represents the contents of the workspace pointer register **73**

LPTR represents the contents of the list pointer register **74**

AREG represents the contents of the A register **71**

BREG represents the contents of the B register **72**

OREG represents the contents of the operand register **65**

A transfer from one register to another is represented by an assignment, eg:

```
BREG:=AREG
```

which means that the contents of the A register are copied to the B register, replacing the previous contents of the B register.

The memory in the transputer is represented by a vector:

```
memory
```

An individual word in memory is identified by subscripting the vector eg:

```
memory [AREG]
```

which means the contents of the word in memory whose address is the contents of the A register.

A transfer between memory and a register is similarly represented by an assignment eg:

```
memory [AREG]:=WPTR
```

which means that the contents of the word in memory whose address is the contents of the A register is replaced by the contents of the workspace pointer register.

Three procedures (PROC) "run", "wait" and "moveto" occur frequently in the following description. They are used in scheduling and will be explained below in connection with scheduling. Meanwhile, they are defined as follows, wherein link [process] represents the contents of the process register **47** of a serial link **25** and NIL represents a special value which is not the workspace pointer of any process. READY represents a further special value used by the serial links:

```
1  PROC run (w)
2    IF
3      w <> READY
4      SEQ
5        memory [LPTR - 2] := w
6        LPTR := w
```

-continued

```
7   w = READY
8   SKIP
1   PROC wait
2   SEQ
3   memory [WPTR − 1 ] := IPTR
4   for each external request from a serial link
5   SEQ
6   run ( link [ process ] )
7   link [ process ] := NIL
8   WPTR := memory [ WPTR − 2 ]
9   IPTR := memory [ WPTR − 1 ]
1   PROC moveto (w)
2   SEQ
3   IF
4   WPTR = LPTR
5   LPTR := w
6   WPTR <> LPTR
7   memory [ w − 2 ] := memory [ WPTR − 2 ]
8   WPTR := w
```

In the above procedures, line numbers have been added for reference purposes in explanation which will be given below.

Function and Operation Definitions

These are now set out below using the notation defined above. These will be further explained below in connection with FIG. 4 and scheduling.

load from workspace(function code 0)

Definition:
```
SEQ
BREG := AREG
AREG := memory [ WPTR + OREG ]
```
Purpose: to load into the A register the value of a location in the current process workspace.

store to workspace(function code 1)

Definition:
```
SEQ
memory [ WPTR + OREG ] := AREG
AREG := BREG
```
Purpose: to store a value in a location in the current process workspace.

load pointer into workspace(function code 2)

Definition:
```
SEQ
BREG := AREG
AREG := WPTR + OREG
```
Purpose: to load into the A register a pointer to a location in the current process workspace
to load a pointer to the first *location of a vector of locations in* the current process workspace.

load from workspace and increment(function code 3)

Definition:
```
SEQ
BREG := AREG
AREG := memory [ WPTR + OREG ]
memory [ WPTR + OREG ] := AREG + 1
```
Purpose: to load into the A register the value of a location in the current process workspace, and increment the location
to facilitate the use of workspace locations as loop counters, incrementing towards zero
to facilitate the use of workspace locations as incrementing pointers to vectors of words or bytes.

load from vector(function code 4)

Definition:
```
AREG := memory [ AREG + OREG ]
```
Purpose: to load into the A register a value from an outer workspace
to load a value from a vector of values
to load a value, using a value as a pointer (indirection) - in this case OREG = 0

store to vector(function code 5)

Definition:
```
SEQ
memory [ BREG + OREG ] := AREG
AREG := BREG
```
Purpose: to store a value in a location in an outer workspace
to store a value in a vector of *values*
to store a value, using a value as a pointer (indirection) - in this case OREG = 0

load literal(function code 6)

Definition:
```
SEQ
BREG := AREG
AREG := OREG
```
Purpose: to load a value

add literal(function code 7)

Definition:
```
AREG := AREG + OREG
```

-continued

Purpose:	to add a value to load a pointer to a location in an outer workspace to load a pointer to a location in a vector of values
jump(function code 8)	
Definition:	IPTR := IPTR + OREG
Purpose:	to transfer control forward or backwards, providing loops, exits from loops, continuation after conditional sections of program
jump non zero(function code 9)	
Definition:	IF AREG <> 0 IPTR := IPTR + OREG AREG = 0 SKIP
Purpose:	to transfer control forwards or backwards only if a non-zero value is loaded, providing conditional execution of sections of program and conditional loop exits to facilitate comparison of a value against a set of values
load pointer into code(function code 10)	
Definition:	SEQ BREG := AREG AREG := IPTR + OREG
Purpose:	to load into the A register the address of an instruction to load the address of a vector of data forming part of the program
call procedure(function code 11)	
Definition:	SEQ memory [WPTR − 1] := IPTR IPTR := AREG AREG := WPTR moveto (WPTR + OREG)
Purpose:	to provide an efficient procedure call mechanism to facilitate code sharing, where two identical procedures are executed on the same processor
Indirect Functions(function code 13) operate	
Definition:	operate (OREG)
Purpose:	perform an operation, using the contents of the operand register (OREG) as the code defining the operation required.
Prefixing Functions prefix(function code 14)	
Definition:	OREG := OREG << 4
Purpose:	to allow instruction operands which are not in the range 0–15 to be represented using one or more prefix instructions
negative prefix(function code 15)	
Definition:	OREG := (NOT OREG) << 4
Purpose:	to allow negative operands to be represented using a single negative prefix instruction followed by zero or more prefix instructions.
Operations(function code 13) reverse(operation code 0)	
Definition:	SEQ OREG := AREG AREG := BREG BREG := OREG
Purpose:	to exchange the contents of the A and B registers to reverse operands of asymmetric operators, where this cannot conveniently be done in a compiler
equal to zero(operation code 1)	
Definition:	IF AREG = 0 AREG := TRUE AREG <> 0 AREG := FALSE

-continued

Purpose:	to test that A holds a non zero value to implement logical (but not bitwise) negation to implement A = 0 as eqz A <> 0 as eqz, eqz if A = 0 . . . as jnz if A <> 0 . . . as eqz, jnz
greater(operation code 2)	
Definition:	IF BREG > AREG AREG := TRUE BREG < = AREG AREG := FALSE
Purpose:	to compare A and B (treating them as twos complement integers), loading −1 (true) if B is greater than A, 0 (false) otherwise to implement B < A by reversing operands to implement B <= A as (gt, eqz), and B >= A by reversing operands and (gt, eqz)
and(operation code 3)	
Definition:	AREG := AREG /\ BREG
Purpose:	to load the bitwise AND of A and B, setting each bit to 1 if the corresponding bits in both A and B are set to 1, 0 otherwise to logically AND two truth values
or(operation code 4)	
Definition:	AREG := BREG \/ AREG
Purpose:	to load the bitwise OR of A and B, setting each bit to 1 if either of the corresponding bits of A and B is set, 0 otherwise to logically OR two truth values
exclusive or(operation code 5)	
Definition:	AREG := BREG >< AREG
Purpose:	to load the bitwise exclusive OR of A and B setting each bit to 1 if the corresponding bits of A and B are different, 0 otherwise to implement bitwise not as (ldl −1, xor)
add(operation code 6)	
Definition:	AREG := BREG + AREG
Purpose:	to load the sum of B and A to compute addresses of words or bytes in vectors
subtract(operation code 7)	
Definition:	AREG := BREG − AREG
Purpose:	to subtract A from B, loading the result to implement A = B as sub, eqz A <> B as sub, eqz, eqz if A = B as sub, jnz, . . . if A <> B as sub, eqz, jnz, . . .
run process(operation code 8)	
Definition:	SEQ memory [AREG − 1] := BREG run (AREG)
Purpose:	to add a process to the end of the active process list
pause(operation code 9)	
Definition:	SEQ run (WPTR) wait ( )
Purpose:	to temporarily stop executing the current process to share the processor time between the processes currently on the active process list
join(operation code 10)	
Definition:	IF memory [AREG] = 0 moveto (memory [AREG + 1])

-continued

	memory [AREG] <> 0 SEQ memory [AREG] : = memory [AREG] − 1 wait ( )
Purpose:	to join two parallel processes; two words are used, one being a counter, the other a pointer to a workspace. When the count reaches 0, the workspace is changed
synchronise(operation code 11)	
Definition:	IF memory [AREG] = NIL SEQ *memory* [AREG] := WPTR wait ( ) memory [AREG] <> NIL SEQ run (memory [AREG]) memory [AREG] := NIL
Purpose:	to allow two processes to synchronise and communicate using a channel
return(operation code 12)	
Definition:	SEQ moveto (AREG) IPTR := memory [WPTR − 1] AREG := BREG
Purpose:	to return from a called procedure
rotate bytes(operation code 13)	
Definition:	AREG := (AREG << 8) \ /(AREG >> (bitsperword - 8))
Purpose:	to rotate the bytes in the A register to allow 8 bit byte values to be combined to form a single word value to allow a word value to be split into several component 8 bit values
shift right(operation code 14)	
Definition:	AREG := AREG >> 1
Purpose:	to shift the contents of the A register one place right
shift left(operation code 15)	
Definition:	AREG := AREG << 1
Purpose:	to shift the contents of the A register one place left

It will be seen that the above function set includes direct functions, indirect functions and prefixing functions. At the start of execution of any instruction, regardless of the function selected for that instruction, the predetermined set of bit positions in the instruction buffer **63** which receive the function part of the instruction are used to provide an input to the decoder **64** whereas the other predetermined bit positions in the instruction buffer **63** which represent the data part of each instruction are used to load the least significant four bit positions of the operand register **65**. If the function is a direct function, the processor then acts in accordance with the selected function on the contents of the operand register **65**. If the function is an indirect function, the contents of the operand register **65** are used to determine the nature of the operation to be carried out and the operation is effected on data held in other registers. At the end of any instruction in which the function is direct or indirect, the operand register **65** is cleared to zero. If the function is a prefix function, the processor operates to transfer existing data in the operand register **65** to positions of higher significance and then load into the vacated positions of lower significance data derived from the data part of the instruction. At the start of each instruction, the instruction pointer is incremented. Consequently the instruction pointer always points to the next instruction to be executed. As mentioned, the instruction pointer IPTR is stored in register **67**.

The operand register **65** is used for several different purposes. The "data" which it receives with each instruction may be a literal value for use in a computation or in the case of an indirect function, it is the definition of the required operation. A further important use is that for some functions, the data value in the operand register **65** will be combined with the data in the workspace pointer register **73** to locate an address where the value of a particular variable is to be found or to be stored. For example, the workspace pointer register **73** will contain the workspace pointer WPTR of the current process. This points to a reference memory address for the workspace. Variables or other points will be defined and stored in that workspace at address locations which are offset by known amounts from the address pointed to by the workspace pointer WPTR. That offset will generally be specified by an instruction portion and stored in operand register **65**. Indeed, the load and store from workspace instructions will implicitly refer to a memory location defined by the combination (illustratively the additive sum) of the contents of WPTR register **73** and the operand register **65**. Furthermore, the contents of the operand register **65** will be combined with the contents of other registers such as the A register **71** or the IPTR register **67**, for accessing vectors or for branching in the program. Examples of this will be given below.

It will be seen that the direct functions are selected to cover the most commonly required actions within the

microcomputer in order to maximise efficiency of operation. By using 4 bits to represent the function element of each instruction, the function set uses codes 0 to 15 although no function has been allocated to code **12**. Code **13** is used to indicate the indirect function which in this case is the "operate" function causing the least significant 4 bits of the instruction to be loaded into the operand register **65** in the usual way but the contents of that operand register are then used by the processor to determine an operation on data held in other registers. It will be appreciated that in this way the number of operations can be extended whilst maintaining uniformity of an 8 bit instruction. By use of the prefix or negative prefix functions before the "operate" instruction, the contents of the operand register **65** can be varied to provide a much greater selection of operations than is set out above. The use of pfix and nfix will be described in more detail below with reference to FIG. **8** but first it is necessary to describe further the operation of the micro-instruction program **13**.

The micro-instruction program is the means of generating control signals which control the switches **56** and inferface control logic **15** (FIG. 3) in order to carry out the required "function" of each sequential instruction arriving in the instruction buffer **63** from the microcomputer program. The micro-instruction program consists of a list of micro-instructions stored in rows and columns in the ROM **13**. The ROM **13** provides an output, called a micro-word, which may consist of 68 bits each providing a control signal and divided up into a plurality of different fields, each field consisting of a predetermined group of bit positions. The output at any one time is provided at selected bit positions depending on the micro-instruction selected. Each field may relate to a specific area of control, such as for example, one field controls which register is connected to the X bus, another field controls which register is connected to the Y bus, another field controls which register is connected to the Z bus, another field controls the action of the ALU **55** and another field controls feed back signals to the multiplexor **9** and MIR **80**. One field controls the interface control logic **15** and provides micro-instruction output signals such as "Read", "Write" and "Next instruction required (NEXT)" to allow the microprogram to control communication between registers and the memory **19** through the interface **14**.

The particular micro-instruction selected in the ROM **13** depends on the address in the MIR **80**, which is a 7 bit register providing a row and column selection in the ROM **13**. At the beginning of each instruction received by the instruction buffer **63** the "function" is decoded by the decoder **64** and is passed through the condition multiplexor **9** to provide an address for selection of the micro-instruction in the ROM **13**. Some functions may require only one micro-instruction to carry out the function, in which case the ROM **13** provide a micro-word output dependent on the address decoded by the decoder **64** and the function is completed in one cycle of operation, herein called a minor cycle, of the ROM **13**. Other functions require a succession of micro-instructions, and therefore minor cycles. In this case, the decoder **64** provides the MIR **80** with an address for the ROM **13** to select the first micro-instruction necessary for that function. Thereafter the microprogram proceeds to execute a sequence of micro-instructions, each taking one minor cycle, and each micro instruction provides in a field of its output micro-word 7 bits for the MIR **80** so as to identify the address of the next micro instruction to be exectuted in the sequence. The least significant two bits of the MIR **80** may be conditionally set, so that the next minor instruction is selected as a result of conditions produced by a previous minor cycle, and fed back through the multiplexor **9** to effect the address in the MIR **80**. This allows the next micro-instruction to be selected from four possible options depending on for example the values in the various registers shown in FIG. **3**. If the two conditional bits of the MIR **80** are not set conditionally then the next micro-instruction indicated by the address in the MIR **80** is unconditionally executed. When all micro-instructions have been executed in order to achieve operation of the instruction in the instruction buffer **63**, the control signal "NEXT" is generated in a field of the micro-word output of the ROM **13**, thereby demanding the next instruction from the memory **19** to the instruction buffer **63**.

Each minor cycle consists of two phases, a source phase and a destination phase. The control signals generated from the ROM **13** fall into three groups; those which are active only during the source phase, those which are active only during the destintion phase and those which are active throughout the whole minor cycle. In order to control the occurrence and duration of the control signals, the timing control is arranged to provide four different strobe signals indicated in FIG. **6**. These are a source strobe **150**, a destination strobe **151**, a column precharge strobe **152** and a micro-instruction strobe **153**. The source strobe is a timing signal which allows a register to place its contents onto a bus and its duration is long enough to allow the arithmetic logic unit to form a result. The destination strobe signals are arranged to allow registers to accept data from a bus. The micro-instruction strobe is used to generate the address of the next micro-instruction from the condition multiplexor **9**. The column precharge strobe is used to precharge the bus lines X and Y to a high state ready for the next source strobe. The relative timing and duration of these strobes is shown in FIG. **6**. They are generated by the arrangement shown in FIG. **7**. The clock pulses from pin **28** (FIG. **1**) generate a GO signal for the beginning of each minor cycle. This signal is passed through four successive delay units within the CPU **12** so that the micro-instruction strobe **153** is derived from the output of the first delay unit **154**, the destination strobe **151** is derived from the output of the second delay unit **155**, the column precharge signal **152** is derived from the output of the third delay unit **156** and the source strobe **150** is derived from the output of the fourth delay unit **157**. The operation of the processor is therefore synchronised to the external clock input **28**.

USE OF VARIABLE LENGTH OPERANDS

As already explained above, the microcomputer is capable of operating with a variable length operand. Although each instruction allocates 4 bit locations to an operand, it is possible to build up in the operand register **65** an operand up to 16 bits by use of the functions pfix and nfix corresponding to codes **14** and **15** in the function set set out above. This operation can best be understood with reference to FIG. **8**. This indicates the operand register **65** having four sections each with 4 bits. The arithmetic logic unit **55** is indicated having four sections corresponding to 4 bits of increasing significance and the connection between the 0 register **65** and the arithmetic logic unit **55** is controlled via a gate **90** selectively controlling transmission through the Y bus

to the arithmetic logic unit. The Y and Z buses are each shown separated into four parts, each handing four bits of data of different significance, e.g. Y[3:0] represents the part of the Y bus handling the four digits of least significance whereas Y[15:12] handles the four digits of greatest significance, and similarly for the Z bus. Each section of the operand register **65** other than the least significant 4 bits, can be supplied through a gate **91** from the Z bus or alternatively it can be fed with a zero from the gate **92**. The instruction from the instruction buffer **63** in FIG. **8** is divided so that the least significant 4 bits are fed to the least significant 4 bit position of the 0 register **65** and the function element is used to select an address in the micro-instruction program **13** as previously described with reference to FIG. **3**. The truth table of FIG. **8** indicates three alternative possibilities where the function corresponds to pfix or nfix or neither. It also lists the corresponding control signals which are fed onto lines **100** to **104** from the micro-word output of the ROM **13**, and the duration of those signals.

The micro-word output control signals used in this case are as follows:

1. OPD NOT O—meaning that the operand register **65** is not supplied with zeroes if the truth table has a "1" but is supplied with zeroes if the truth table has a "0".

2. NEXT—meaning that the operand register **65** will be loaded with the next operand from the instruction buffer **63** if the truth table has a "1" but not if the truth table has a "0".

3. Y FROM OPD—meaning that the Y bus receives the operand from the operand register **65** if the truth table has a "1" but not if the truth table has "0".

4. Z FROM Y—meaning that the Z bus output from the ALU **55** will receive data from the Y bus if the truth table has a "1", but not if the truth table has a "0".

5. Z FROM NOT Y—meaning that the ALU **55** will cause the signal on the Y bus to be inserted and passed to the Z bus if the truth table has a "1" but not if the truth table has an "0".

The duration of these five control signals in each minor cycle is indicated in FIG. **8** wherein S indicates duration in the source phase only, D indicates duration only in the destination phase and S+D indicates duration in both.

The micro-word control signal on line **100** operates the gates **91** and **92** to allow the Z bus to unload into the operand register **65** in response to the functions pfix and nfix whereas any other function causes the three most significant stages of the operand register **65** to be zeroed by an input through the gate **92**. All instructions generate the control signal NEXT on the last minor cycle and this is applied to line **101** to cause the operand register **65** to be loaded with the next operand. Line **102** receives the signal "Y FROM OPD" and causes the operand register to be connected through the gate **90** to the Y bus for both pfix and nfix. Line **113** receives the control signal "Z FROM Y" and causes the arithmetic logic unit **55** to transmit to the Z bus the signal on the Y bus for pfix but not for nfix. Line **104** receives "Z FROM NOT Y" and allows the signal on Y to be inverted and supplied through the ALU **55** to the Z bus for nfix but not for pfix. The signals on lines **100**, **103** and **104** exist throughout the source and destination phases of each minor cycle whereas the signal on line **101** exists only in the destination phase and the signal on line **102** exists only in the source phase. When the function is pfix, it can be seen that signals corresponding to a truth condition are supplied on lines **100**, **101**, **102** and **103** and in this way, the 4 bits of operand in the least significant section of the operand register **65** are advanced through the arithmetic logic unit to the next significant stage of the operand register **65** thereby allowing a further 4 bits of operand to be loaded into the least significant positions of the operand register **65**. This operation is repeated each time an instruction is derived with pfix function up till a maximum of 16 bits of operand. Similarly if the function is nfix, the process is generally similar in allowing each successive 4 bits of operand to be moved up into a higher stage of the 0 register **65** without zeroes being written in after each instruction. This allows a negative operand to be built up to a maximum of 16 bits. The truth table indicates that if the function is neither pfix nor nfix, the control signal on line **100** causes zeroes to be fed into the three upper significant stages of the 0 register **65** (representing bits15 to 4)at the end of that instruction.

SCHEDULING OF PROCESSES

As already indicated, the microcomputer may operate a number of concurrent processes. It therefore provides a system of scheduling to determine which process shall be performed at any particular time. At any one time the WPTR register **73** (FIGS. **3** and **4**) holds the workspace pointer of the process currently being executed. However the workspace of the current process and the workspaces of other processes waiting to be executed form a list in which one location of each workspace holds the workspace pointer of the next process on the list. Another location in each process workspace holds the instruction pointer identifying the next instruction which is to be carried out for that process when it becomes the current process. Furthermore the LPTR register **74** contains the address of the workspace for the last process currently waiting to be executed. In this way new processes can be added to the end of the list and the LPTR register **74** always indicates the current end of the list. The processor normally executes the processes on the list in sequence only advancing to a subsequent process when the current process executes a "pause" operation (code **9** in the operations list) or when the current process deschedules itself by executing a "join" operation (code **10** in the operations list) or a synchronise operation (code **11** in the operations list). In any of those situations, the current process ceases to carry out further instructions and the processor save the instruction pointer IPTR in the process workspace as indicated at **36** in FIG. **2**, and moves onto the next process which has been identified by the address of the next process, shown as **35** in FIG. **2** and then loads into the IPTR register **67** the IPTR for the new process. So that there is always at least one process running, a null process is provided and the null process is run when no other process is active.

The procedures "run", "wait", and "moveto" defined above are used in scheduling. A process will be "scheduled" when it is the current process or is on the linked list of processes which are waiting to be executed. A process becomes "descheduled" when it is taken off the linked list. A descheduled process will never be executed unless some other process or instruction schedules it, i.e. adds it to the end of the linked list. It will be recalled that LPTR register **74** (FIG. **3**) is used to store the workspace pointer for the last process on the list. Hence, it must be adjusted whenever a process is added to the linked list. Also, when a process is to be sched-

uled, the CPU 12 must be able to determine which instruction is to be executed next for the process. This is done by storing in memory the appropriate instruction pointer IPTR, which is in IPTR register 67 while the process is current. Such storage is done, for example at memory location 36 (FIG. 2).

In describing these procedures, it will be convenient to refer to FIG. 4 which illustrates workspaces 32 and 33 more particularly, as well as registers 65, 67, 71, 73 and 74. FIG. 4 shows representative memory addresses and contents of the workspaces.

The process which has the workspace 32 is made the current process by inserting its workspace pointer WPTR into register 73. In this case, WPTR equals 10000. When the process becomes the current process the processor finds the next instruction to be executed by examining WPTR-1, i.e. the contents at memory location 9999, to find a pointer 84 to an instruction and loads this pointer in the IPTR register 67. While this is the current process, the processor will use the contents of IPTR register 67 to point to the next instruction.

During the processing, it will use variables whose addresses are formed by combining a reference value, such as the WPTR or the contentsof the A register 71, and an operand placed in register 65. In a load from workspace operation an operand of "2" will refer to whatever is at memory location 10002 while the process corresponding to workspace 32 is current. When processing is to stop, the linked list is consulted. Elements 85 and 86 are part of the linked list. The processor will look at WPTR-2 to find WPTR 85 at memory location 9998, pointing to the next workspace. Pointer 85 will contain the number 11000 which points to workspace 33. If the process corresponding to workspace 33 is the last process on the linked list, the LPTR register 74 will contain the pointer 11000. No pointer 86 will be stored at memory location 10998 until some process is added to the linked list.

Turning now to the three procedures, PROC run (w) is used the schedule a process defined by w i.e., add it to the linked list. This procedure has been defined above and reference will now be made to that definition and the line numbers used in the definition.

If the value of w is the special value "READY" no action is performed. Further explanation of this will follow later with reference to communications between different microcomputers. Otherwise w is a pointer to a process workspace, and lines 5 and 6 will be executed in sequence. In line 5, LPTR means the contents of LPTR register 74, which is a pointer to the reference address for the workspace for the last process on the linked list. The memory whose address is LPTR-2 would contain the address of the workspace pointer for the next process, but as yet there is none because LPTR corresponds to the last process. Line 5 now assigns w (the workspace pointer in the process w) to memory location LPTR-2, so process w is now at the end of the linked list. At this point, the contents of LPTR register 74 points not to the last process w, but to the penultimate process. This is corrected in line 6 which enters into LPTR register 74 the workspace pointer for process w. Because of this step, further processes can be added to the linked list without deleting process w unintentionally, which would happen if LPTR register 74 were not updated. With reference to FIG. 4, if there are only two processes scheduled, as shown, and process w corresponds to a workspace whose pointer is 12000, PROC run (w) would enter 12000 in memory [10998] and enter 12000 into register 74.

The procedure called "wait" can be used alone or in combination with PROC run (w). By itself, PROC wait deschedules the current process and enables the system to execute the next scheduled process, executing it where appropriate in its program instead of at its first instruction. In sequence with PROC run (w), PROC wait causes the microcomputer to stop the current process, schedule it at the end of the list of processes to be executed, and proceed to the next scheduled process. Reference will now be made to the previous definition of PROC wait. When procedure "wait" is called (line 1), a sequence is commenced (line 2) having four steps (lines 3, 4, 8 and 9). Lines 4–7 relate to external requests, and discussion of this can be deferred, although link [process] represents the contents of process register 47 of serial link 25 (FIG. 2). In line 3, memory [WPTR-1] is the memory space at the address WPTR-1, which is based on the reference address WPTR of the current process. That memory location is, in the preferred embodiment, used to point to the next instruction to be executed when the process is recommenced. The contents of IPTR register 67 always points to the instruction to be executed next for the current process. Hence, line 3 simply stores in memory (preferably on-chip) the pointer to the next instruction to be executed when, if ever, the process being descheduled becomes current. Assume that the current process is process w. If the procedure PROC run (w) has preceded PROC wait, then at this time, the current process (w) will have been added at the end of the linked list (by PROC run (w)), LPTR register 74 will have been updated (also by PROC run (w)), and now the pointer to the next instruction for process w will have been stored at a known location, memory [WPTR-1], with respect to the workspace pointer address (WPTR) for process w. Thus, process w is ready now to be deactivated. Line 8 of PROC wait looks to the linked list for the next process. Its workspace will be pointed to by the contents at address WPTR-2 of the current workspace w. Hence, line 8 of PROC wait assigns to WPTR register 73 the workspace pointer for the next process on the linked list. Now the reference address WPTR has advanced, and the system next finds out what the next instruction is for this next process by looking at the pointer stored at the memory whose address is WPTR-1. To use FIG. 4, consider that workspace 32 is current and its process receives an instruction which includes PROC wait. Initially, WPTR is 10000. At line 8, register 73 is set to the contents found at memory address 9998, which will be the pointer 11000. At line 9, register 67 is set with the instruction pointer found at memory address 10999. Thus, if PROC run (w) is followed by PROC wait, the current process is added to the end of the list (its workspace pointer is stored on the linked list), the pointer to its next instruction is stored in memory, it is deactivated, and the next process on the linked list is commenced beginning at the proper instruction. All of this is done using only four registers. This arrangement permits the scheduling and descheduling of processes which are limited in number by only the amount of memory in the system.

The procedure named "moveto" can be used to set the workspace reference pointer to a different address in the workspace for the current process, without necessarily changing to a new IPTR. Thus, if a process has its reference workspace pointer at 10000, moveto (10200)

could be used to set the registers to change the reference pointer to **10200** for this same process. This will be described as follows with reference to the previous definition of PROC moveto (w). Line **2** of the definition declares this a sequence of steps. Lines **3** and **8** are equally offset from the left margin, so they both will be done in sequence. Assume that the system is not on the last process. Hence, line **4** will be false, so the system will jump to line **6**. The condition at line **6** will be true, so line **7** will be executed. Line **7** sets the contents at memory addres w-2 to the workspace pointer for the next process on the linked list. Next, line **8** changes the contents of the WPTR register **73** to the value w. Now register **73** points to a new reference address for the current process. At the customary offset (minus 2) from this new reference address will be found a pointer to the workspace for the next process to be scheduled. In the event that there is no next process, then line **4** will be true and LPTR register **74** will have its contents adjusted to point to w as the reference address for the last process (line **5**), after which the register **73** for holding a pointer to the reference address of the current process will be adjusted to point to w.

Having now described FIG. **4** with reference to scheduling, some functions and operations will be further described with reference to FIG. **4**.

load from workspace

The load from workspace function (function code **0**) copies the contents at a specific memory location and puts it implicitly into the A register. This function and configuration of the preferred embodiment implicitly refers also to the memory whose address is defined by an offset from the current workspace pointer which serves as a reference. This reference address is always stored in the WPTR register **73**, and the offset is contained in the operand register **65**. The expression, "memory [WPTR+OREG]" therefore refers to the contents of the memory whose address is found by adding (summing) the contents of WPTR register **73** and register **65**. A "load" refers to the A register **71**, and the contents of the stack will be shifted down by one register, i.e. the contents of the A register will be shifted into the B register (to make room for the data to be loaded into AREG), and the contents of BREG will be loaded into the C register, if any. With reference to FIG. **4**, if WPTR is **10000**, then "load from workspace" using codes **0 2** will mean load variable **2** into the A register.

store to work space

This "store to workspace" function (function code **1**) implicitly means whatever is in the A register **71** into the memory space whose address is offset from the reference address (contained in WPTR register **73**) by the offset contained in the operand register **65**. Also, the stack moves up (BREG moves into AREG, and CREG moves into BREG). Referring to FIG. **4** if WPTR=**10000** and OREG=1, then this function means store the contents of the A register **71** into memory location **10001**, which is the location for storing variable **1**.

load pointer into workspace

The function "load pointer into workspace" (function code **2**) does not store any data into the workspace. Instead, it load the A register **71** with a pointer to a particular location in workspace. This will be used, for example, in connection with the "load from vector" instruction which references a particular portion of a vector which can be stored in the workspace. Thus, referring to FIG. **4** a workspace **32** will be referred to by the workspace points WPTR which is **10000**. At a known location within the workspace, there can be a vector. The vector will have a plurality of locations such as **10200**, **10201** and **10202**. The beginning of the vector will be a particular offset (**200**) away from the workspace pointer (**10000**). Thus, to find the beginning of the vector, the offset (**200**) will be loaded into the operand register **65** and then the instruction "load pointer into workspace" will add these two numbers to obtain a sum 10200 which is an address. This function will place the address **10200** into the A register, which point to the beginning of the vector. Thereafter, the "load from vector" operation will be used to find particular memory locations with respect to the beginning of the vector, and therefore it uses the offset in the operand register **65** but in combination with the A register **71** instead of the workspace pointer register **73**.

load literal

The "load literal" function (function code **6**) literally loads whatever is in the operand register **65** into the A register **71** (the top of the evaluation stack). With respect to FIG. **5**, the last four bits of any given instruction will be loaded into the operand register **65**, but by use of the prefixing functions, more than 4 bits can be stored in the operand register Illustratively, however, an instruction having the codes in decimal notation of **6 13** has two parts, a function part and a data part, as explained referring to FIG. **5**. The first number "**6**" is the function code, indicating that this is a "load literal" function. The second part of the instruction is the data value "**13**". Accordingly this instruction "**6 13**" would mean load the number **13** into the A register **71** and shift the previous contents of the A register into the B register **72**.

jump

The "jump" function (function code **8**) is used for branching in a program. The instruction to be executed next by the processor for the current process is pointed to by the contents of the IPTR register **67** which contains the instruction pointer. The jump instruction adds the contents of the operand register **65** to the instruction pointer. Through use of the prefixing functions, the instruction pointer can have values added to it or subtracted from it, to jump forward or backward in a program.

call procedure

The "call procedure" function (function code **11**) uses the "moveto" procedure which was described above. "Call procedure" first stores IPTR in memory at the customary location for the instruction next to be executed (e.g. memory location **9999** in FIG. **4**). Next it transfers into the instruction pointer register **67** the contents of the A register **71** which will have been loaded with the pointer to an instruction next to be executed after the "call procedure" function is completed. Then the A register **71** is loaded with the workspace pointer. Following this, the "moveto" procedute changes the reference pointer WPTR so that usually it points to a different address in the current workspace. It will be remembered to "moveto ( )" procedure will set the contents of the WPTR register **73** to whatever is within the parenthesis following the word "moveto". Thus, after a "call procedure," the system now has the workspace pointer pointing to a different location within the same workspace for the current process and is prepared to execute a different instruction which was previously contained in the A register **71**. The converse

operation is effected by use of the RETURN operation (operation code 12).

run process

This operation 37 run process" (operation code 8) is generally used in the creation of a process which will have its own workspace and set of instructions. The A register 61 will have been loaded with a workspace pointer for the workspace for the new process, and the B register 72 will have been loaded with a suitable instruction pointer for the new process. Operation "run process" stores the instruction pointer in memory at the proper offset from the workspace pointer, and then it calls the procedure PROC run ( ), discussed above, using the workspace pointer in the parentheses. As discussed, this will schedule the new process, i.e. it will add the new process to the linked list.

pause

The "pause operation (operation code 9) appears in a program to prevent any single process from using the ALU 55 to the exclusion of other processes. This operation is inserted into loops by the compiler. This operation adds the current process to the end of the linked list, stores the necessary pointers, causes the current process to cease execution for the time being, and makes the next process on the linked list the current process. The contents of the evaluation stack are not preserved because "pause" is executed at a time when such contents can be discarded without harming the process.

join

This "join" operation (operation code 10) is used for example when there are concurrent processes, and it is intended that they should all be at a point in the program at the same time. Consider an original process P(0) which at a certain point in the program spreads into n concurrent subprocesses P(1), P(2), P(3) . . . P(n). When these are done, a final process P(n+1) is to execute. However, such final process should not occur until all of P(1) . . . P(n) have terminated. The "join" operation is used for this. A counter is set up in the workspace, and the A register 71 points to the memory location where the count is stored. The count corresponds to the number of subprocesses (which are still active (not terminated). Each subprocess ends with a "join" operation. After a subprocess reaches its "join" operation, it checks the count. If the count is zero, then the program moves to the final process using the "moveto" procedure. If the count is not zero, the count is decremented by one count, and then the subprocess is caused to "wait" as described above. The other subprocesses are executed until zero count is reached.

synchronise

The "synchronise" operation (operation code 11) is quite important to concurrent processing, for its use assures that two processes will be at the same point at some time. This will be discussed further in connection with FIG. 9 and the discussion entitled, "Communication Between Processes On the Same Microcomputer." Briefly however, if two processes X and Y on the same chip wish to communicate, presumably because one process is computing data which the other process needs, a channel 40, 41, 42 or 43 (FIG. 2) is used. Each process will have a "synchronise" operation. The first process to reach its "sync" operation will look at the channel. The channel address will have been loaded into the A register 71, so "memory [AREG]" refers to the channel. The expression "NIL" in the definition of this operation refers to a predetermined datum recognised as a nil. If NIL is found in a first part of the channel by the first process to reach its sync operation, such process will place its workspace pointer into the first part of the channel and then will deschedule itself. Assume that it is process X which first reaches "sync". Process X now waits for process Y to reach its "sync" operation. When this happens, process Y will check the first part of the same channel, and it will not find NIL but will instead find the workspace pointer for process X. In response, it schedules process X (adds it to the end of the linked list). The first part of the channel returns to NIL. Generally there will be at least a second part to the channel where data for transfer from one process to the other will be placed. Also, synchronise operations generally occur in pairs. The first "sync" operations in two processes can cause the process to wait for data and then transfer it when it is ready. The second "sync" instructions cause acknowledgments. Thus, a process which is inputting data from a process will "sync". If the data is not ready, it will "wait". When the data is ready by the supply process, that supplying process will schedule the receiving process, which will then take the data. Then "sync" instructions by each acknowledge the transfer. The first "sync" by the process supplying the data will indicate that the data is ready to be taken.

COMMUNICATION BETWEEN PROCESSES ON THE SAME MICROCOMPUTER

As already explained, the microcomputer permits communication between processes which may be on the same microcomputer or on different microcomputers. For example, one process may be the measurement of distance travelled by a motor car and a second process the measurement of consumption of fuel relative to distance travelled for that vehicle. The first process may receive as an input, data representating rotations of the vehicle wheel and provide an output representing miles travelled. The second process may receive as an input data relating to fuel quantity consumed but it also needs to communicate with the first process to derive information about distance travelled before it can provide a useful output regarding fuel consumption relative to distance. In the case of process to process communications on the same microcomputer communication is carried out in this example through the channels 40 to 43 indicated on FIG. 2. This operation involves the use of the synchronise operation, this requires a program instruction consisting of function code 13 and operation code 11 from the above list of functions and operations. Each channel 40 to 43 consists of two consecutive word locations in memory, one providing a "process location" and the other a "data location". The channel is a unidirectional communication channel which is shared by two and only two processes at any one time. When an active process x wishes to communicate with a process y on the same microcomputer, it follows a sequence which will be described with reference to FIGS. 9*a* to 9*e*. Firstly, process x identifies the address of the channel (marked 40) and loads the data it wishes to communicate into the data location of the channel. It also executes an instruction for a synchronise operation. Provided the process location of channel 40 does not already have the workspace pointer of the process y awaiting to receive the data, the synchronise operation causes the work space pointer of process x to be recorded in the process location of channel 40 and uses a "wait" procedure of deschedule process x. This is the position shown in FIG. 9*b*. In FIG. 9, the work space pointer of process X is referred to as "X" and the data

being communicated is referred to as "DATA." Process x now waits until process y is ready to receive the data. When process y wishes to receive the data it carries out an instruction for a synchronise operation to see if the communication channel **40** is ready to transmit data. In carrying out this instruction, process y locates the workspace pointer "X" of process x in the process location of channel **40** and as can be seen from the synchronise operation set out in the list of operations, the execution of a synchronise operation causes a "run" procedure to remove the workspace pointer of process x from channel **40** and add process x to the end of the list of processes waiting to be executed. This is the position of FIG. **9***c*. Process y then reads the data from the data location of channel **40** and then operates a further instruction for a synchronise operation to indicate that it has received the data. This loads the workspace pointer "y" of process y into the process location of channel **40** and causes process y to wait. This deschedules process y leaving the channel **40** in the condition shown in FIG. **9***d*. Once the list on which process x is waiting reaches process x so that process x is reactivated, it performs a further instruction for a synchronise operation which now locates the workspace pointer "Y" of process y in the process location of channel **40** and this allows process x to continue to be operated. At the same time it causes a "run" procedure on process y so that process y is again added to the end of the waiting list of processes and is ready to run. The communication channel **40** is then in the condition shown in FIG. **9***e* with process x continuing the process y waiting on the list. In this way, synchronisation of communication is achieved by both processes operating a "handshake" operation in which both processes execute two instructions for synchronise operations one of which deschedules the process and that descheduled process is only put back onto the list when an appropriate signal has been received from the other of the communicating processes.

A specific example of programs and instruction sequences necessary to carry out two communicating processes on the same microcomputer will now be described with reference to FIG. **10**. This illustrates the two processes referred to above for measuring miles travelled and fuel consumption of a motor vehicle. The microcomputer **170** has in its memory space a first workspace **171** for the first process which is counting the variable "rotations" and a second workspace **172** for the second process which is counting the variable "miles". Workspace **171** has a word location **173** containing the address of the input channel **174** called "rotation" which forms part of a serial link arranged to receive a message for each wheel revolution from an external revolution detector (not shown). The workspace **171** has a further word location **175** containing the address of a two word memory channel **176** called channel "mile" which in this case receives an output from the process of workspace **171** indicating 1 mile of travel for each 1000 revolutions of the vehicle wheel.

For this first process the program using OCCAM language is as follows:

```
1.  VAR rotations:
2.  WHILE TRUE
3.  SEQ
4.  rotations:= 0
5.  WHILE rotations < 1000
6.  SEQ
7.  rotation ? ANY
8.  rotations:= rotations +1
9.  mile ! ANY
```

Line numbers are not part of the program but have been added to facilitate explanation. Line 1 declares a variable to exist; it is called "rotations". Line 2 is an endless loop because the condition TRUE is always true. Start with zero rotations (line 4). Line 7 waits for any input from the channel named "rotation." When one is received, the variable "rotations" is incremented by one. Eventually there will have been 1000 rotations, and Line 5 will be false. Lines 6, 7 and 8 will then be skipped and Line 9 will output a datum to the channel named "mile".

The compiler will convert these OCCAM statements to the following machine instructions:

	Instruction Sequence					
				Function code	Data	Program in OCCAM language
						VAR rotations: WHILE TRUE SEQ
1.	L1:					
2.		ldl	0	6	0	rotations := 0
3.		stw	0	1	0	
4.	L2:					WHILE rotations <1000 SEQ
5.		ldw	0	0	0	
6.		pfix		14	3	
7.		pfix		14	14	
8.		ldl	1000	6	8	
9.		opr	gt	13	2	
10.		jnz	L3	9	9	
11.		ldw	1	0	1	rotation ? ANY
12.		opr	sync	13	11	
13.		ldw	1	0	1	
14.		opr	sync	13	11	
15.		ldw	0	0	0	rotations:=rotations +1
16.		adl	1	7	1	
17.		stw	0	1	0	
18.		opr	pause	13	9	
19.		nfix		15	0	
20.		j	L2	8	0	
21.	L3:					
22.		ldw	2	0	2	mile ! ANY
23.		opr	sync	13	11	
24.		ldw	2	0	2	
25.		opr	sync	13	11	
26.		opr	pause	13	9	
27.		nfix		15	2	
28.		j	L1	8	7	

Once again, line numbers have been added for explanatory purposes only. Lines 1, 4 and 21 are simply reference locations in the program. Line 2 loads the value 0 into A register **71**. Line 3 stores the data in the A register into workspace. Because the data part of the instruction is 0, there is no offset from the reference address for this process. Thus, the workspace pointer register **73** now contains a workspace pointer WPTR which points to a reference address in memory where 0 is stored for the variable "rotations". Line **5** loads the A register **71** from workspace. Because the data portion of the instruction (which would be loaded into operand register **65**) is 0, the offset from the reference address WPTR of the workspace is 0. In lines 6, 7 and 8 the decimal value 1000 is to be added. This requires a prefixing operation because 1000 cannot be represented using four binary bits in the data portion of the instruction. Thus, function

code **14** (pfix) is used. The decimal number 1000 in binary is 1111101000. Because this requires ten bits, and the data portion of standard instructions is four bits, three steps are required to load this value into the operand register. Line 8 includes the code for the "load literal" function, so at this time, the A register **71** will be loaded with the binary value of 1000. This causes the transfer of the contents of the A register (which are 0) to the B register **72**.

Line 9 calls for an indirect function, the operation "greater than". This causes a comparison of the A and B registers. Unless the B register contents are greater than the A register contents, this operation will result in FALSE (**0**).

Line 10 is the "jump nonzero" operation. If the results of line 9's operation were true, then the A register would be set to a nonzero value, and line 10 would cause a jump of 9 lines forward, indicated by the number "9" in the data part of the code. This should jump the program ahead to line 21, the output portion. Assuming that 1000 rotations have not yet been counted, line 11 is next executed. This load from workspace function has an operand of +1, which means the offset from the reference address is +1. At this memory address will be found the address of the channel named "rotation" and this address will be loaded from the workspace into the A register **71**. Line 12 causes a synchronise operation. Line 13 again loads the address of the channel "rotation" and line 14 again synchronises to complete the input operation. In this simple example, no data is transferred. Line 15 now loads the variable which is in workspace, offset 0, into the A register, i.e. loads the current value of "rotations" into the A register. Line 16 literally adds the data value 1 to the contents of the A register. Line 17 stores the contents of the A register in the workspace at an offset equal to 0. Hence, the variable "rotations" has now been incremented in response to receipt of data from the channel "rotation". Line 18 is a pause operation which allows the next process to be executed, adding this present process to the end of the list. Note that at this point in the program, the contents of the A register **71** and B register **72** are not relevant to the process. Lines 19 and 20 execute a jump backwards using the negative prefix function. Line 22 loads the contents of the workspace which is offset 2 locations from the reference location. This will be the address of the channel named "mile" and it will be loaded into the A register. A sync operation is performed at line 23. The output is completed by a further "sync" which occurs at line 25. At line 26 another pause is inserted to cause the next process to schedule and to add this process to the end of the linked list. At lines 27 and 28, a jump backwards is executed using negative prefixing. The second process relating to the variable "miles" uses the workspace **172**. The workspace **172** has a word location **177** containing the address for the "mile" channel **176** which is used to provide an input to the process of workspace **172**. A further word location **178** has the address of a second input which in this case is a channel **179** called channel "fuel" forming part of a serial link arranged to receive a message from an external fuel gauge (not shown) each time a gallon of fuel is consumed. The workspace **172** has a further word location **180** having the address of an output channel **181** called channel "consumption" forming part of a serial link arranged to output the distance travelled while the last gallon of fuel was consumed. Clearly the process in workspace **172** needs to communicate with the process in workspace **171** in order to obtain via channel **176** messages indicating the number of miles travelled. The instruction sequence and program from the process in workspace **172** are as follows:

Instruction sequence					
			Function code	Data	Program in above defined OCCAM
					VAR miles:
					SEQ
L1:					
	ldl	0	6	0	miles := 0
	stw	0	1	0	WHILE TRUE
L2:					ALT
	ldw	1	0	1	mile ? ANY
	ldv	0	4	0	
	opr	eqz	13	1	
	jnz	13	9	9	
	ldw	1	0	1	
	opr	sync	13	11	
	ldw	1	0	1	
	opr	sync	13	11	
	ldw	0	0	0	miles := miles + 1
	adl	1	7	1	
	stw	0	1	0	
	pfix		14	1	
	j	L4	8	0	
L3:					
	ldw	2	0	2	fuel ? ANY
	ldv	0	4	0	
	opr	eqz	13	1	
	jnz	L4	9	12	
	ldw	2	0	2	
	opr	sync	13	11	
	ldw	2	0	2	
	opr	sync	13	11	
					SEQ
	ldw	3	0	3	consumption ! miles
	ldw	0	0	0	
	stv	1	5	1	
	opr	sync	13	11	
	ldw	3	0	3	
	opr	sync	13	11	
	ldl	0	6	0	miles := 0
	stw	0	1	0	
L4:					
	opr	pause	13	9	
	nfix		15	1	
	j	L2	8	0	

COMMUNICATION BETWEEN PROCESSES ON DIFFERENT MICROCOMPUTERS

A network of interconnected microcomputers is shown in FIG. **11** in which four microcomputers are illustrated. It will be understood that the network may be extended in two or three dimensions as required. Each of the microcomputers is of similar structure and is interconnected with the serial link of another microcomputer by two unidirectional wires **185** and **186** each of which extends between the output pin **27** on one microcomputer and the input pin **26** of another microcomputer. The wires **185** and **186** are each used solely for these two pin to pin connections and are not shared by other microcomputers or memory connections. Communication between processes in different microcomputers is effected in generally similar manner using an identical sequence of synchronise operations and this will be described with reference to FIGS. **2**, **11**, **12** and **13**. In place of the channel **40** (FIG. **2**), a serial link has an input channel **45** and an output channel **46** each consisting of a process register **47** and data register **48** which can be addressed in the same way as the word locations for the memory channels **40** to **43**. They are

however operated by control logic **50** which will be described further with reference to FIGS. **15** and **16**. In FIG. **11**, an output channel is shown with a data register **187** and a process register **188**. An input channel is shown having a process register **189** and a data register **190**. The control logic shown in FIG. **2** is not shown in FIG. **11**, but it will be understood that such logic is present.

When data is transmitted through serial links between two microcomputers, it is in the form of a series of data strings transmitted serially in the form of packets as shown in FIGS. **13***a* and **13***b*. A data packet is transmitted by an output pin **27** to an input pin **26** and has the form shown in FIG. **13***a*. It starts with two successive bits of value 1 followed by 16 data bits and a final stop bit of value 0. An acknowledge packet, as shown in FIG. **13***b* is sent from the output pin **27** of a microcomputer receiving a data packet to the input pin **26** of the microcomputer which sent the data packet. The acknowledge packet consists of a start bit of value one followed by a stop bit of value 0. The output control logic of each serial link arranges for each output pin **27** to transmit bits of value 0 continuously when it is not sending data or acknowledge packets and consequently the input control logic ignores all signals on the input pins **26** until it receives a "1" start bit of a packet.

When the process register **47** (FIG. **2**) of the input or output channel **45**, **46** holds the workspace pointer (WPTR) of a process, the control logic **50** is able to generate requests (called input or output requests) to the CPU (**12**) for the CPU **12** to schedule the process by adding its workspace pointer to the list awaiting execution. The sync logic **10** provides a selector which is used by the CPU **12** to examine each of the request signals from the serial links in turn. Whenever an active process is descheduled by execution of the "wait" procedure, the CPU **12** looks to see if there are any requests from a serial link. If there are several external requests, the CPU **12** services all of them in sequence before executing the next process on the list. The CPU **12** services any requests by scheduling the process held in the process register of the channel which generated the request, the resetting the process register **47** to NIL. The process register **47** of the input or output channels in a link **25** contains the special value READY when that channel is ready to perform communication. The sync operation will cause the procedure "run" which detects the special value READY and instead of scheduling a process, activates the control logic **50** in the link. The control logic in a link may perform a synchronise operation on a channel. The synchronise operation tests the process location of the channel. If the value is NIL, it replaces the value with the special value READY and waits until a sync operation caused by a process instruction on the process register resets the value to NIL. Otherwise, it generates a request to the CPU **12** to schedule the process in the process register as described above, and the CPU then resets the value of the process register to NIL. As a result, a process may use the sync operation to synchronise with the control logic **50** in a link **25** in the same way as it is used to synchronise with another process.

The output control logic **50** in a link **25** first synchronises with a process using the process register in the output channel, when transmits data in data packets from the data register in the output channel via the output pin **27** (FIGS. **2** and **11**), then waits for an acknowledge packet signal on the input pin **26**, then synchonises with the process again using the process register in the output channel. The output control logic **50** performs this operation repeatedly. The input control logic in a link first waits for data from the input pin **26** to arrive in the data register in the input channel, then synchronises with a process using the process register in the input channel, then synchronises again with the process using the process register in the input channel, then transmits the acknowledge packet signal to the output pin **27**. The input control logic performs this operation repeatedly.

In the following, it is assumed that a process x operated by microcomputer **1** in FIG. **11** wishes to output data through a serial link to a process y operated by microcomputer **2**. To effect this output, the process x stores the data to be output in the data register **187** of the output channel and executes a sync operation on the process register **188** to cause the serial link to start transmission of the data through the pin **27**. The process then executes a further sync operation on the same process register **188** to wait until an acknowledge packet is received through the input pin **26** of microcomputer **1**. The acknowledge packet signifies that the process y operated by microcomputer **2** has input the data. To input, the process y executes a sync operation on the process register **189** of the input channel of microcomputer **2** to wait for the data packet to arrive from the pin **26** of the microcomputer **2**. It then takes the data from the data register **190** and executes a further sync operation to cause the acknowledge signal to be transmitted from the output pin **27** of microcomputer **2**.

FIG. **12** shows sequentially the contents of the process registers **188** and **189** during a typical sequence of operations occurring when the process x and y communicate via the serial link. Reference numerals **188***a–e* represent successive states of the contents of the process register **188** and reference numerals **189***a–e* similarly represent successive states of the contents of the process register **189**. First, process x addresses the output channel of microcomputer **1** and loads the data to be output to the data register **187** and performs a sync operation on the output process register **188**. Assuming that the process register **188** contains the special value READY **188***a*, indicating that the serial link is ready to output, the sync operation resets the value of the process register **188** to NIL **188***b*. As a result the control logic causes the data from the data register **187** to be transmitted via the single wire connection **185** to the input data register **190** in the microcomputer **2**. Provided that process y is not yet waiting for the input, the control logic in microcomputer **2** changes the value of the process register **189** from NIL **189***a* to READY **189***b*, indicating that the data has been received. Process y then executes a sync operation on the process register **189**, which has the effect of changing the value of the process register from READY **189***b* to NIL **189***c*. Assuming that microcomputer **2** is ready to transmit an acknowledge signal to microcomputer **1**, the control logic changes the value of process register **189** back to READY **189***d*. Process y then takes the data from the data register **190** of the input channel and executes a further sync operation on the process register **189**. This resets the process register **189** to NIL **189***e*. As a result the control logic transmits an acknowledge signal through the single wire connection **186**. This acknowledge signal is received by the input pin **26** of the microcomputer **1** operating process x. Assuming that process x executes a second sync operation before the acknowledge signal is received, process

x is descheduled by the procedure "wait", and its workspace pointer "X" is stored in the process register **188** (**188***c*). When the acknowledge packet is received the control logic of the serial link generates a request to the CPU of microcomputer **1** to schedule process x. This request is serviced by the CPU of microcomputer **1** as soon as the current process is descheduled and the CPU adds process x to the end of the list and resets the process register to NIL (**188***d*). The control logic now resets the process register to READY (**188***e*), thereby indicating that the link is ready for a further output. The state of the serial links is now the same as it was before the communication took place, as shown in the sequence of FIG. **12**, ready for the next communication. FIG. **14** illustrates the operation on two separate microcomputers of the processes previously described with reference to FIG. **11**. In this case however the workspace **171** for counting rotations is on a microcomputer **191** whereas the workspace **172** for counting miles is on a separate microcomputer **192**. The two microcomputers **191** and **192** are interconnected through respective serial links **25**. Similar reference numerals are used in FIGS. **14** and **10** for similar parts. The only change is that channel "mile" **176** in FIG. **10** is replaced in FIG. **14** by a channel "mile" **176***a* forming an output channel of a serial link in microcomputer **191** and channel "mile" **176***b* forming an input channel of a serial link in microcomputer **192**. The sequence of instructions and program used to operate the two processes in FIG. **14** are generally similar to those already described for FIG. **10** except that the address of channel "mile" used by each of the processes will now be the address of a channel of a serial link rather than a channel in memory.

DESCRIPTION OF LINK CONTROL LOGIC

The control logic **50** (FIG. **2**) for each of the input and output channels of the serial links will now be described in further detail with reference to FIGS. **15** and **16** in which FIG. **15** shows the control logic for the output channel **46** and FIG. **16** shows the control logic for the input channel **45**.

To output, the control logic **50** (FIG. **2**) of a link first synchronises with a process using the output process register **47** (FIG. **15**), then transmits the data from the output data register **48** to the pin **27**, then waits for the acknowledge signal from the pin **26**, then synchronises with a process again using the output process register **47**. The control logic **50** performs this operation repeatedly.

To input, the control logic **50** (FIG. **2**) of a link first waits for data to arrive from the input pin **26** and transfers it to the input data register **48**, then synchronises with a process using the input process register **47** (FIG. **16**), then synchronises again with the process using the input process register, then tranmits the acknowledge signal to the pin **27**. The control logic **50** performs this operation repeatedly.

The values taken by the output and input process registers **47** may be NIL indicating that neither a process nor the control logic is waiting to synchronise, READY indicating that the control logic is waiting to synchronise, or it may be the workspace pointer of a process waiting to synchronise.

In a link, each process register **47** and each data register **48** is connected to the bus **16** through an address decoder **193**. The bus **16** incorporates signal lines for the address, data, and control. Control includes a "write" signal, a "read" signal and a "busy" signal. The "busy" signal is used to ensure that both the CPU and the link control logic do not attempt to change the value of the process register simultaneously.

Each process register **47** in a link incorporates logic **194** to detect if the value in the process register is READY, NIL or a workspace pointer.

The output data register **48** (FIG. **15**) is connected to the output pin **27** through an AND gate **195** and an OR gate **196**. The input data register **48** (FIG. **16**) is connected directly to the input pin **26**.

Associated with each process register in a link is a request latch **197** which may be tested by the CPU. Whenever the CPU performs a WAIT procedure, the state of all request latches is tested. If a request latch is set, the process whose workspace pointer is held in the corresponding process register is scheduled by adding its workspace pointer to the end of the list. The request latch is cleared whenever the CPU writes to the process register.

The input and output of data through the link is controlled by four state machines **282**, **283**, **284** and **285**. Each state machine consists of a state register to hold the current state, and a programmable logic array. The programmable logic array responds to the value of the state register and the input signals to the state machine, and produces a predetermined pattern of output signals and a new value for the state register. A counter **286** is used to count bits as they are transmitted through the link, and a further counter **287** is used to count bits as they are received through the link.

The input and output channel control and data state machines have the following inputs and outputs, wherein the name of the input or output indicates the purpose of the signal.

OUTPUT CONTROL STATE MACHINE 285

(FIG. 15) reference numeral	signal name	purpose
inputs:		
200	Mbusy	Memory bus busy
201	Reset	Transputer reset
202	Pregready	Process Register = READY
203	Pregnil	Process Register = NIL
204	Pregwptr	Process Register holds a workspace pointer
205	Datagone	Data transmitted from output data register
264	Ackready	Acknowledge received by input state machine
outputs:		
210	Setrequest	Set cpu request
211	Datago	Initiate data transmission
212	SetPregready	Set Process Register to READY
213	SetPregnil	Set Process Register to NIL
265	Acktaken	Confirm receipt of acknowledge

OUTPUT DATA STATE MACHINE 284

(FIG. 15) reference numeral	signal name	purpose
inputs:		
201	Reset	Transputer reset
211	Datago	Initiate data transmission
220	Countzero	Test if bit count zero
261	Ackgo	Initiate acknowledge transmission
outputs:		
221	Loadcount	Set Bit Counter to number of bits to be transmitted

-continued

OUTPUT DATA STATE MACHINE 284

(FIG. 15) reference numeral	signal name	purpose
222	Deccount	Decrease bit counter by one
223	Oneout	Set output pin to one
224	Dataout	Set output pin to least significant bit of shift register
225	Shiftout	Shift data register one place
205	Datagone	Transmission of data complete
260	Ackgone	Transmission of acknowledge complete

INPUT CONTROL STATE MACHINE 283

(FIG. 16) reference numeral	signal name	purpose
inputs:		
200	Mbusy	Memory bus busy
201	Reset	Transputer reset
262	Dataready	Data received from pin
242	Pregready	Process Register = READY
243	Pregnil	Process Register = NIL
244	Pregwptr	Process Register holds a workspace pointer
260	Ackgone	Transmission of acknowledge complete
outputs:		
220	Setrequest	Set cpu request
222	SetPregready	Set Process Register to READY
222	SetPregnil	Set Process Register to NIL
261	Ackgo	Initiate acknowledge transmission
263	Datataken	Confirm receipt of data

INPUT DATA STATE MACHINE 282

(FIG. 16) reference numeral	signal name	purpose
inputs:		
201	Reset	Transputer reset
230	Datain	Data from pin
231	Countzero	Test if bit count zero
outputs:		
240	Loadcount	Set Bit Counter to number of bits to be received
241	Deccount	Decrease bit counter by one
244	Shiftin	Shift data register one place taking least significant bit from pin
245	Setdataready	Reception of data complete
246	Setackready	Reception of acknowledge complete

The sequences of each state machine are set out below with reference to present state, next state, input and output of each machine.

In any state, the outputs listed under the "outputs" column are one, and all other outputs are zero. All inputs are ignored except those mentioned in the "inputs" column. The symbols $\wedge$, $\vee$ and Δ are used to denote the boolean operations and, or and not respectively.

OUTPUT CONTROL STATE MACHINE 285

State	Inputs	Outputs	Next state
any	Reset	SetPregnil	sync1
sync1	Mbusy		sync1
sync1	(ΔMbusy) / \ Pregnil	SetPregready	syncreq1
sync1	(ΔMbusy) / \ Pregwptr	Setrequest	syncreq1
syncreq1	ΔPregnil		syncreq1
syncreq1	Pregnil		send1
send1	ΔDatagone	Datago	send1
send1	Datagone		send2
send2	Datagone		send2
send2	ΔDatagone		waitack1
waitack1	ΔAckready		waitack1
waitack1	Ackready		waitack2
waitack2	Ackready	Acktaken	waitack2
waitack2	ΔAckready		sync2
sync2	Mbusy		sync2
sync2	(ΔMbusy) / \ Pregnil	SetPregready	syncreq2
sync2	(ΔMbusy) / \ Pregwptr	Setrequest	syncreq2
syncreq2	ΔPregnil		syncreq2
syncreq2	Pregnil		sync1

OUTPUT DATA STATE MACHINE 284

State	Inputs	Outputs	Next state
any	Reset		idle
idle	(ΔDatago) / \ (ΔAckgo)		idle
idle	Ackgo	Oneout	ackflag
idle	(ΔAckgo) / \ Datago	Oneout	dataflag
ackflag			ackend
dataflag		Oneout Loadcount	databits
databits	ΔCountzero	DecCount Shiftout Dataout	databits
databits	Countzero		dataend
dataend	Datago	Datagone	dataend
dataend	ΔDatago		idle
ackend	Ackgo	Ackgone	ackend
ackend	ΔAckgo		idle

INPUT CONTROL STATE MACHINE 283

State	Inputs	Outputs	Next state
any	Reset	SetPregnil	receive1
receive1	ΔDataready		receive1
receive1	Dataready		sync1
sync1	Mbusy		sync1
sync1	(ΔMbusy) / \ Pregnil	SetPregready	syncreq1
sync1	(ΔMbusy) / \ Pregwptr	Setrequest	syncreq1
syncreq1	ΔPregnil		syncreq1
syncreq1	Pregnil		sync2
sync2	Mbusy		sync2
sync2	(ΔMbusy) / \ Pregnil	SetPregready	syncreq2
sync2	(ΔMbusy) / \ Pregwptr	Setrequest	syncreq2
syncreq2	ΔPregnil		syncreq2
syncreq2	Pregnil		receive2
receive2	Dataready	Datataken	receive2
receive2	ΔDataready		acksend1
acksend1	ΔAckgone	Ackgo	acksend1
acksend1	Ackgone		acksend2
acksend2	Ackgone		acksend2
acksend2	ΔAckgone		receive1

INPUT DATA STATE MACHINE 282

State	Inputs	Outputs	Next state
any	Reset		idle
idle	ΔDatain		idle
idle	Datain		start
start	ΔDatain	SetAckready	idle
start	Datain	LoadCount	databits
databits	ΔCountzero	Shiftin DecCount	databits
databits	Countzero	Shiftin	dataend
dataend	—	SetDataready	idle

As shown in FIG. **16**, the input control logic includes a flip-flop **280** connected to the output **246** of the input data state machine **282**. A further flip-flop **281** is connected to the output **245** of the input data state machine **282**. Both control state machines are controlled by clock pulses derived from the clock **28**. For some of the links, both data state machines are also controlled by clock pulses derived from the clock **28**. For the link shown in FIGS. **15** and **16**, the data state machines are controlled by clock pulses derived from a different clock **22** related in phase to clock **28**, which allows this link to operate at a lower speed. Two different clock frequencies can be obtained in order to achieve maximum efficiency depending on the type of microcomputer network which is operated. When microcomputers are grouped closely together communications between them can be carried out more quickly in which case a higher clock frequency can be used. A lower clock frequency can be used to enable satisfactory communication where the microcomputers are more remote and require a lower operating speed.

In both the input and output channels the control state machine monitors the content of the process register **47** and when appropriate generates a CPU request on line **199** by setting the latch **197**.

The output control state machine **285** first synchronises with a process using the output process register **47**. It then uses the "datago" signal **211** to cause the output data state machine **282** to output the data in the output data register **48** through the pin **27**. The output data state machine **284** sends the data in the manner described with reference to FIG. **13***a* and shifts the data in the register **48** until a count in the counter **286** expires. When it has done this it returns the "datagone" signal **205** to the output control state machine to indicate that the transfer of data is complete and that the "datago" signal should be removed. The output control state machine then waits for the "ackready" signal **264** from the latch **280**, signifying that the input data state machine **282** has received an acknowledge packet as described in FIG. **13***b* from the pin **26**. In response to the "ackready" signal **264**, the output control state machine outputs an "acktaken" signal **265**, which resets the latch **280**. The output control logic then uses the output process register **47** to synchronise again with the outputting process.

The input data state machine **282** and the microcomputer at the other end of the link is waiting for "start bit" to appear on the input pin **26**. When a data packet is detected, the input data state machine **282** of that microcomputer shifts data into the data shift register **48** until the counter **287** indicates that the appropriate number of bits have been received, and then sets the "data received" latch **281**. The input control state machine **283** detects the "dataready" signal **262** and responds by resetting the "data received" latch **281**. It then synchronises with an inputting process using the input process register **47**. It then synchronises again with the inputting process using the process register **47** to confirm that the process has taken the data from the data register **48**, and then uses the "ackgo" signal **261** to cause the output data state machine to transmit an acknowledge packet via the pin **27**. When the output data state machine **284** is not transmitting data it generates the start and stop bits which constitute the acknowledge packet described in FIG. **13***b*. The input data state machine **282** of the microcomputer which transmitted the data packet detects the acknowledge packet and sets the "acknowledge received" latch **280**. As described above, the output control state machine **285** of the transmitting microcomputer has been waiting for this and on detecting the signal resets the latch **280** and performs a second synchronise operation. The state of the link logic in both the output and input links is now the same as it was before the communication took place so that it is ready for the next transmission.

CHIP AND MEMORY FORMATION

As mentioned above, the microcomputer of this example is particularly advantageous in having sufficient memory in the form of RAM on the chip (integrated circuit device) to enable the microcomputer to operate without the necessity for external memory, although external memory can be used when required. There are a number of problems in providing sufficient space for adequate memory on the same chip as the processor. It is necessary to minimise the area required for each memory cell as well as reducing noise interference in the RAM from asynchronously operating circuitry such as a processor on the same chip, while at the same time providing a satisfactory manufacturing yield of acceptable microcomputers from a number of silicon chips, particularly as the memory may be the largest and densest component created on the chip.

In order to minimise the chip area required for each memory cell, this example uses static RAM cells (SRAM) using high impedance resistive load rather than the more conventional depletion transistor loads or complementary pull-up transistors. The manufacturing technology used in this example employs a film of high resistivity polycrystalline silicon in which the resistive loads are formed. The memory may have 32K bits of SRAM where each cell consists of transistors having gates formed in a film of polycrystalline silicon. The transistor gates and resistive loads may be formed in the same, or different films of polycrystalline silicon.

Resistor load SRAMs are susceptible to interference from electrical noise injected into the silicon material in which they are formed and stored data can be corrupted by any minority carriers which may be present. In order to shield the SRAM from noise generated by other on chip circuitry and from minority carriers injected by other on chip circuitry the SRAM is formed in an electrically isolated area of silicon as shown in FIG. **17**. An n-channel substrate **300** is formed with separate p-wells **301**, and **302**. The arry of RAM cells are isolated from other circuitry and associated substrate noise by locating the RAM array in the p-well marked **301**. This isolates the RAM cells from minority carriers generated in the substrate by the well-to-substrate potential barrier and any minority carriers generated within the well have a high probability of being collected in the substrate. In FIG. **17**, the RAM array will be an n-channel

array located in the p-well **301**. Any n-channel transistors of peripheral circuitry are isolated from the RAM array by placing them in a further p-well **302**.

This technique is fully compatible with either NMOS or P-well CMOS manufacturing technology. In the current example P-well CMOS is used and any p-channel transistors of peripheral circuitry are placed on the n-substrate and isolated from the RAM array by the well-to-substrate potential barrier. Each well containing a memory array is surrounded by a metal ground which contacts the memory array well around its periphery via a heavily doped p diffusion. Within the memory array there is a p diffusion contacting the well to ground for each pair of cells. Substrate bias is unnecessary.

In order to provide acceptable manufacturing yield of products from silicon chips, memory redundancy is incorporated. The memory is divided into rows and columns accessible respectively by row and column decoders. In addition to the normal rows and normal columns the redundancy provides some additional rows and columns together with spare row and column decoders in order to obtain access to the spare rows and columns. The spare column decoders and spare row decoders each incorporate fuses which for example can be open circuited by use of a laser so that when any defective rows or columns are determined during test, fuses can be open circuited by laser techniques to disable the row or column decoder of any normal rows or columns which have been found to be defective and the replacement row or column from the redundant rows and columns can be brought into an enabled position by programming the appropriate spare row decoder or spare column decoder with the address of the defective row or column.

In order to allow N-well CMOS manufacturing technology to be used the following alternative isolation technique may be employed. Referring to FIG. **18** a low resistivity P type substrate (**405**) is used on which a high resistivity P type epitaxial layer is formed.

The cell array is formed in this epitaxial layer in region (**401**) and is entirely surrounded by a deep N-well diffusion (**402**). Minority carriers generated by other circuitry in region (**403**) will be attracted to the N-wells (**402**) where they become harmless majority carriers, or will recombine in the heavily doped P-type substrate (**405**). P-channel transistors are placed in N-wells (**404**) where they are isolated by the well to substrate potential barrier.

ADDITIONAL MATERIAL

The invention is not limited to the details of the foregoing example. For instance, although the serial links shown in FIG. **2** have separate process registers **47**, the function provided by the registers **47** may be effected by memory locations in the RAM **19**. In this case the CPU must be able to identify the serial link which it is serving and this may be achieved by connecting each channel of each serial link separately to the sync logic **10** in FIG. **2**.

One set of data registers and buses is shown in FIG. **3** and in some cases it may be desirable to include two such sets in one microcomputer, or even to have two CPUs in one microcomputer.

The principle described above of using pfix and nfix functions to vary the length of operand is applicable to a microcomputer of any word length.

The invention is not limited to a machine operating with 16 bit words nor to 16 bit operand registers, e.g. processors having a word length of 8 bits or multiples of 8 bits may use these instructions. The invention is particularly applicable to 32 bit word microcomputers.

The CPU may include further registers, in an evaluation stack, such as a CREG or even DREG in addition to the A and B registers. Some functions and operations may then be modified to allow the additional registers. For example:

```
Areg := Breg may be replaced by SEQ
Areg := Breg
Breg := Creg
Creg := Dreg
Breg := Areg may be replaced by SEQ
Dreg := Creg
Creg := Breg
Breg := Areg
```

Other functions or operations may of course be added to exploit the extra registers. Although the illustrated embodiment described herein and shown in FIG. **3** includes only an A register and a B register, in a preferred embodiment of the present invention, three registers are used in a stack.

It will be appreciated that in the above description, the function set lists a plurality of functions followed by an extendable list of operations which may be selected by use of the indirect function "operate". In all cases these functions and operations can be considered as forms of instruction usable in the program to operate the microcomputer. However in order to obtan the advantages discussed above for a fixed format of "instruction" as shown in FIG. **5**, the list of functions and operations can be considered as a set of primary instructions (consisting of the direct functions, prefixing functions and indirect functions) and a set of secondary instructions (consisting of the operations which may be selected by use of the indirect function). To maximise efficiency, the primary instructions which are most commonly used require only 4 bits of the instruction format shown in FIG. **5** and so the other 4 bits can be used for data to be loaded into the operand register **65** and used as an operand for the instructions. For the secondary instructions which are less commonly used, all 8 bits of the instruction format shown in FIG. **5** are needed to identify the instruction required. Consequently the fixed format of the instruction shown in FIG. **5** allows no data to accompany a secondary instruction and secondary instructions therefore operate on data held in registers other than the operand register **65**.

Although the instruction format shown in FIG. **4** comprises 8 bits divided into two halves, it will be understood that other bit lengths may be used and the division into function and data need not necessarily provide equal bit lengths for the two parts.

It is to be appreciated that the present arrangement described herein provides a combination which dramatically improves the efficiency and throughput of the microcomputer. By using instructions having a constant format, by having a function set where the most often used functions are directly available whereas other functions are indirectly available, by arranging for communication between processes and synchronisation among them, by permitting point-to-point communication between microcomputers, and by providing memory on the same chip as each microprocessor, a microcomputer according to various aspects of the inven-

tion can achieve a speed of 10 million instructions per second. An array housed on a board of only 10 inches by 20 inches should be able to achieve a speed of 1000 million instructions per second. A Transputer (trade mark) microcomputer array using OCCAM (trade mark) should be able to achieve speeds approximately two orders of magnitude faster, than, for example, a Motorola 68000 using PASCAL. A single Transputer programmed in OCCAM should be about two or three times faster than a single 68000 microprocessor using PASCAL. In the prior art, when microcomputers are added in an array, the incremental gain in performance is progressively less and less with an increase in processors. However, by using the microcomputer of this example, the increase in performance is a linear function of the number of processors. Thus it will be appreciated that the present combination achieves dramatically increased performance over the state of the art.

We claim:

1. A network of interconnected microcomputers, each microcomputer comprising:

(a) a single integrated circuit chip having a substrate of semiconductor material of a first type,

(b) an on-chip high density RAM array having at least one K byte for holding a program containing instructions for execution by said on-chip processor,

(c) a plurality of communication links each forming an interconnection with an adjacent microcomputer in the network,

(d) an instruction pointer circuit for addressing said RAM to obtain program instructions therefrom,

(e) an instruction receiving circuit coupled to said RAM for receiving said instructions from said program stored in said RAM,

(f) an instruction decoder circuit coupled to said receiving circuit for decoding instructions received by said instruction receiving circuit,

(g) a plurality of on-chip transistors comprising circuitry operable independently of the operation of said RAM,

(h) first isolation well means formed in said substrate of a semiconductor material of different type than said substrate, said first isolation well means containing all of said memory cells of said high density RAM array, and

(i) second isolation well means separate from said first isolation well means and formed in said substrate of a semiconductor material of different type than said substrate, said second isolation well means containing some of said transistors which are operable independently of said operation of said RAM,

whereby each microcomputer in the network operates in accordance with instructions from program in its on-chip RAM and each on-chip RAM is protected from noise due to operation of independently operating transistors.

2. A microcomputer comprising an on-chip processor and on-chip memory on a single integrated circuit chip having a substrate of semiconductor material of a first type, wherein said on-chip memory comprises a high density RAM array having at least 1K bytes for holding a program containing instructions for execution by said on-chip processor, said microcomputer including:

(a) an instruction pointer circuit for addressing said RAM to obtain program instructions therefrom,

(b) an instruction receiving circuit coupled to said RAM for receiving said instructions from said program stored in said RAM,

(c) an instruction decoder circuit coupled to said instruction receiving circuit for decoding instructions received by said instruction receiving circuit,

(d) a plurality of on-chip transistors comprising circuitry operable independently of the operation of said RAM,

(e) first isolation well means formed in said substrate of a semiconductor material of different type than said substrate, said first isolation well means containing all of said memory cells of said high density RAM array, and

(f) second isolation well means separate from said first isolation well means and formed in said substrate of a semiconductor material of different type than said substrate, said second isolation well means containing some of said transistors which are operable independently of said operation of said RAM,

whereby said high density RAM is located on the same chip as independently operating transistors and is protected from noise due to independent operation of said transistors.

3. A microcomputer according to claim 2 wherein said memory provides at least four K bytes of RAM.

4. A microcomputer according to claim 3 wherein said memory comprises a plurality of RAM cells formed with high impedance resistive loads and transistors.

5. A microcomputer according to claim 4 on which said resistive loads are formed in a film of polycrystalline silicon.

6. A microcomputer according to claim 2 comprising a CMOS structure having an n-type substrate with one or more isolation wells of p-type semiconductor.

7. A microcomputer according to claim 6 in which said memory cells include n-channel transistors located within said p-type well or wells.

8. A microcomputer according to claim 2 having a substrate of low resistivity p-type semiconductor on which a high resistivity p-type epitaxial layer is located, said memory cells being located within said epitaxial layer and surrounded by an n-type region to isolate the memory cells.

9. A microcomputer according to claim 2 in which said memory array comprises a main memory array and a redundant memory array, together with means for enabling use of redundant memory if defective memory elements occur in said main memory array.

10. A microcomputer according to claim 9 in which said redundant memory array incorporates redundant rows and columns of memory elements interconnectable with said main memory array through fuse elements.

11. A microcomputer according to claim 2 comprising a single silicon chip on which is located said processor and further comprising communication channels, said programmable RAM together with said communication channels permitting message transmission to or from a process executed by said processor.

12. A microcomputer according to claim 11 wherein said communication channels include communication links permitting process to process communication with other microcomputers.

13. A microcomputer according to claim 12 further comprising control means for said processor responsive to functions selected from a function set which include

data transfer between registers, between memory and registers, and which enable synchronization of message transfer through said communication channels.

14. A microcomputer according to claim **2** wherein said processor executes a sequence of instructions each one byte long and each having the same format of bit positions, thereby reducing the chip area required by the processor, said registers each having a bit length which is an integral number of bytes.

15. A microcomputer according to claim **2** in which said program is recorded in said memory on the same integrated circuit chip as the processor.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below: Title Page:

Correct the title to read --MICROCOMPUTER WITH HIGH DENSITY RAM IN SEPARATE ISOLATION WELL ON SINGLE CHIP--.

Column 11, lines 23 through 28 delete existing text and insert in its place --

```
SEQ
  P1
  P2
  P3
  ...                                                    --
```

Column 11, lines 36 through 40, delete existing text and insert in its place --

```
SEQ
  in ? x
  x := x + 1
  out ! x
                                                         --
```

Column 11, lines 45 through 50, delete existing text and insert in its place --

```
PAR
  P1
  P2
  P3
  ...                                                    --
```

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 2 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 11, lines 57 through 61, delete existing text and insert in its place --

```
PAR
  in ? x
  out ! y
```

--

Column 12, lines 5 through 13, delete existing text and insert in its place --

```
IF
 condition 1
   P1
 condition 2
   P2
 condition 3
   P3
...
```

--

Column 12, lines 20 through 25 delete existing text and insert in its place --

```
IF
 x >= 0
    y := y+1
 x < 0
    SKIP
```

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 3 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 12, lines 30 through 38 delete existing text and insert in its place --

```
ALT
  input 1
    P1
  input 2
    P2
  input 3
    P3
...                                                    --
```

Column 12, lines 48 through 55 delete existing text and insert in its place --

```
ALT
  count ? ANY
    counter := counter + 1
  total ? ANY
    SEQ
      out ! counter
      counter :=0                                      --
```

Column 13, lines 1 through 3, delete existing text and insert in its place --

```
WHILE x > 5
  x := x - 5                                           --
```

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 4 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 13, lines 15 through 18 delete existing text and insert in its place --

```
VAR v :
P                                                    --
```

Column 13, lines 25 through 27 delete existing text and insert in its place --

```
PROC square ( n, sqr )
  sqr := n * n
```

Column 14, lines 63 through 70; column 15, lines 1 through 15; and column 16, lines 1 to 6, delete existing text and insert in its place --

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 5 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

```
1          PROC run (w)
2            IF
3              w <> READY
4               SEQ
5                 memory [ LPTR - 2 ] := w
6                 LPTR := w
7              w = READY
8               SKIP

1          PROC wait
2            SEQ
3              memory [ WPTR - 1 ] := IPTR
4              for each external request from a serial link
5               SEQ
6                 run ( link [ process ] )
7                 link [ process ] := NIL
8               WPTR := memory [ WPTR - 2 ]
9               IPTR := memory [ WPTR - 1 ]

1          PROC moveto (w)
2            SEQ
3              IF
4               WPTR = LPTR
5                 LPTR := w
6               WPTR <> LPTR
7                 memory [ w - 2 ] := memory [ WPTR - 2 ]
8              WPTR := w                                --
```

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Columns 15, lines 17 through 70 and Columns 17-22 lines 1 through 39, delete existing text and insert in its place --

load from workspace (function code 0)

Definition:
```
SEQ
  BREG := AREG
  AREG := memory [ WPTR + OREG ]
```

Purpose: to load into the A register the value of a location in the current process workspace.

store to workspace (function code 1)

Definition:
```
SEQ
  memory [ WPTR + OREG ] := AREG
  AREG := BREG
```

Purpose: to store a value in a location in the current process workspace

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

load pointer into workspace (function code 2)

Definition:
```
SEQ
  BREG := AREG
  AREG := WPTR + OREG
```

Purpose: to load into the A register a pointer to a location in the current process workspace

to load a pointer to the first location of a vector of locations in the current process workspace.

load from workspace and increment (function code 3)

Definition:
```
SEQ
  BREG := AREG
  AREG := memory [ WPTR + OREG ]
  memory [ WPTR + OREG ] := AREG + 1
```

Purpose: to load into the A register the value of a location in the current process workspace, and increment the location

to facilitate the use of workspace locations as loop counters, incrementing towards zero

to facilitate the use of workspace locations as incrementing pointers to vectors of words or bytes

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 8 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

load from vector (function code 4)

Definition: AREG := memory [AREG + OREG]

Purpose: to load into the A register a value from an outer workspace

to load a value from a vector of values

to load a value, using a value as a pointer (indirection) - in this case OREG = 0

store to vector (function code 5)

Definition:
```
SEQ
  memory [ BREG + OREG ] := AREG
  AREG := BREG
```

Purpose: to store a value in a location in an outer workspace

to store a value in a vector of values

to store a value, using a value as a pointer (indirection) - in this case OREG = 0

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 9 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

load literal (function code 6)

Definition:
```
SEQ
  BREG := AREG
  AREG := OREG
```

Purpose: to load a value

add literal (function code 7)

Definition: AREG := AREG + OREG

Purpose: to add a value

to load a pointer to a location in an outer workspace

to load a pointer to a location in a vector of values

jump (function code 8)

Definition: IPTR := IPTR + OREG

Purpose: to transfer control forward or backwards, providing loops, exits from loops, continuation after conditional selections of program

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

jump non zero (function code 9)

Definition:
```
IF
 AREG <> 0
   IPTR := IPTR + OREG
 AREG = 0
   SKIP
```

Purpose: to transfer control forwards or backwards only if a non-zero value is loaded, providing conditional execution of sections of program and conditional loop exists

to facilitate comparison of a value against a set of values

load pointer into code (function code 10)

Definition:
```
SEQ
  BREG := AREG
  AREG := IPTR + OREG
```

Purpose: to load into the A register the address of an instruction to load the address of a vector of data forming part of the program

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 11 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

call procedure (function code 11)

Definition:
```
SEQ
  memory [ WPTR - 1 ] := IPTR
  IPTR := AREG
  AREG := WPTR
  moveto (WPTR + OREG)
```

Purpose: to provide an efficient procedure call mechanism

to facilitate code sharing, where two identical procedures are executed on the same processor

Indirect Functions (function code 13)

operate

Definition: operate (OREG)

Purpose: perform an operation, using the contents of the operand register (OREG) as the code defining the operation required.

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 12 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Prefixing Functions

prefix (function code 14)

Definition: OREG := OREG << 4

Purpose: to allow instruction operands which are not in the range 0 - 15 to be represented using one or more prefix instructions

negative prefix (function code 15)

Definition: OREG := (NOT OREG) << 4

Purpose: to allow negative operands to be represented using a single negative prefix instruction followed by zero or more prefix instructions.

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 13 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Operations (function code 13)

reverse (operation code 0)

Definition:
```
SEQ
  OREG := AREG
  AREG := BREG
  BREG := OREG
```

Purpose: to exchange the contents of the A and B registers

to reverse operands of asymmetric operators, where this cannot conveniently be done in a compiler

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 14 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

equal to zero (operation code 1)

Definition:

```
IF
 AREG = 0
   AREG := TRUE
 AREG <> 0
   AREG := FALSE
```

Purpose: to test that A holds a non zero value

to implement logical (but not bitwise) negation

to implement

```
A = 0              as eqz
A <> 0             as eqz,  eqz
if A = 0 ...       as jnz
if A <> 0 ...      as eqz, jnz
```

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

greater (operation code 2)

Definition:

```
IF
 BREG > AREG
   AREG := TRUE
 BREG < = AREG
   AREG := FALSE
```

Purpose: to compare A and B (treating them as twos complement integers), loading -1 (true) if B is greater than A, 0 (false) otherwise

to implement B < A by reversing operands

to implement B <= A as (gt, eqz), and B >= A by reversing operands and (gt, eqz)

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 16 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

and (operation code 3)

Definition: AREG := AREG / \ BREG

Purpose: to load to bitwise AND of A and B, setting each bit to 1 if the corresponding bits in both A and B are set to 1, 0 otherwise

to logically AND two truth values

or (operation code 4)

Definition: AREG := BREG \ / AREG

Purpose: to load the bitwise OR of A and B, setting each bit to 1 if either of the corresponding bits of A and B is set, 0 otherwise

to logically OR two truth values

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 17 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

exclusive or (operation code 5)

Definition: AREG := BREG >< AREG

Purpose: to load the bitwise exclusive OR of A and B setting each bit to 1 if the corresponding bits of A and B are different, 0 otherwise

to implement bitwise not as (ldl -1, xor)

add (operation code 6)

Definition: AREG := BREG + AREG

Purpose: to load the sum of B and A

to compute addresses of words or bytes in vectors

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

subtract (operation code 7)

Definition: AREG := BREG - AREG

Purpose: to subtract A from B, loading the result

to implement

A = B	as sub, eqz
A <> B	as sub, eqz, eqz
if A = B	as sub, jnz,...
if A <> B	as sub, eqz, jnz,...

run process (operation code 8)

Definition:
```
SEQ
  memory [ AREG - 1 ] := BREG
  run ( AREG )
```

Purpose: to add a process to the end of the active process list

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 19 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

pause (operation code 9)

Definition:
```
SEQ
  run ( WPTR )
  wait (  )
```

Purpose: to temporarily stop executing the current process

to share the processor time between the processes currently on the active process list

join (operation code 10)

Definition:
```
IF
 memory [ AREG ] = 0
   moveto ( memory [ AREG + 1 ] )
 memory [ AREG ] <> 0
   SEQ
     memory [ AREG : = memory [ AREG ] - 1
     wait (  )
```

Purpose: to join two parallel processes; two words are used, one being a counter, the other a pointer to a workspace. When the count reaches 0, the workspace is changed

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 20 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

synchronize (operation code 11)

Definition:
```
IF
 memory [ AREG ] = NIL
   SEQ
     memory [ AREG ] := WPTR
     wait (  )
 memory [ AREG ] <> NIL
   SEQ
     run ( memory [ AREG ] )
     memory [ AREG ] := NIL
```

Purpose: to allow two processes to synchronize and communicate using a channel

return (operation code 12)

Definition:
```
SEQ
  move to ( AREG )
  IPTR := memory [ WPTR - 1 ]
  AREG := BREG
```

Purpose: to return from a called procedure

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

rotate bytes (operation code 13)

Definition: AREG := (AREG << 8) \/ (AREG >> (bitsperword - 8))

Purpose: to rotate the bytes in the A register
to allow 8 bit byte values to be combined
to form a single word value
to allow a word value to be split
into several component 8 bit values

shift right (operation code 14)

Definition: AREG := AREG >> 1

Purpose: to shift the contents of the A
register one place right

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 22 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

shift left (operation code 15)

Definition: AREG := AREG << 1

Purpose: to shift the contents of the A register one place left --

Column 33, lines 63 through 70, and column 34, lines 1 to 4, delete the existing text and insert in its place --

```
1.   VAR rotations:
2.   WHILE TRUE
3.     SEQ
4.       rotations:= 0
5.       WHILE rotations < 1000
6.         SEQ
7.           rotation ? ANY
8.           rotations := rotations +1
9.       mile ! ANY                                    --
```

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 34, lines 20 through 50, delete existing text and insert in its place --

Instruction Sequence					Program in OCCAM Language
		Function Code		Data	
					VAR rotations:
					WHILE TRUE
					SEQ
1.	L1:				
2.	ldl	0	6	0	rotations := 0
3.	stw	0	1	0	
4.	L2:				WHILE rotations <1000
					SEQ
5.	ldw	0	0	0	
6.	pfix		14	3	
7.	pfix		14	14	
8.	ldl	1000	6	8	

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 24 of 26
DATED : July 14, 1987
INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Instruction Sequence			Function Code	Data	Program in OCCAM Language
9.	opr	gt	13	2	
10.	jnz	L3	9	9	
11.	ldw	1	0	1	rotation ? ANY
12.	opr	sync	13	11	
13.	ldw	1	0	1	
14.	opr	sync	13	11	
15.	ldw	0	0	0	rotations:=rotations + 1
16.	adl	1	7	1	
17.	stw	0	1	0	
18.	opr	pause	13	9	
19.	nfix		15	0	
20.	j	L2	8	0	
21. L3:					
22.	ldw	2	0	2	mile ! ANY
23.	opr	sync	13	11	
24.	ldw	2	0	2	
25.	opr	sync	13	11	
26.	opr	pause	13	9	
27.	nfix		15	2	
28.	j	L1	8	7	--

Column 36, lines 7 through 43, delete existing text and insert in its place --

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698 Page 25 of 26

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Instruction Sequence				Program in above defined OCCAM Language
	Function Code		Data	
				VAR miles:
				SEQ
L1:				
ldl	0	6	0	miles := 0
stw	0	1	0	WHILE TRUE
L2:				ALT
ldw	1	0	1	mile ? ANY
ldv	0	4	0	
opr	eqz	13	1	
jnz	13	9	9	
ldw	1	0	1	
opr	sync	13	11	
ldw	1	0	1	
opr	sync	13	11	
ldw	0	0	0	miles := miles + 1
adl	1	7	1	
stw	0	1	0	
pfix		14	1	
j	L4	8	0	
L3:				
ldw	2	0	2	fuel ? ANY
ldv	0	4	0	
opr	eqz	13	1	
jnz	L4	9	12	
ldw	2	0	2	

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 4,680,698

DATED : July 14, 1987

INVENTOR(S) : May, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Instruction Sequence		Function Code	Data	Program in above defined OCCAM Language
	opr	sync 13	11	
	ldw	2 0	2	
	opr	sync 13	11	
				SEQ
	ldw	3 0	3	consumption ! miles
	ldw	0 0	0	
	stv	1 5	1	
	opr	sync 13	11	
	ldw	3 0	3	
	opr	sync 13	11	
	ldl	0 6	0	miles := 0
	stw	0 1	0	
L4:				
	opr	pause 13	9	
	nifx	15	1	
	j	L2 8	0	--

Signed and Sealed this

Fifteenth Day of November, 1988

Attest:

Attesting Officer

DONALD J. QUIGG

Commissioner of Patents and Trademarks

Exhibit O

DS

UNITED STATES DEPARTMENT OF COMMERCE
Patent and Trademark Office

Address : COMMISSIONER OF PATENTS AND TRADEMARKS
Washington, D.C. 20231

SERIAL NUMBER	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.
07/389,334	08/03/89	MOORE C	A50412WEH

EXAMINER
ENG, D

ART UNIT	PAPER NUMBER
2315	8

DATE MAILED: 12/31/92

FLEHR, HOHBACH, TEST,
ALBRITTON & HERBERT
SUITE 3400
FOUR EMBARCADERO CENTER
SAN FRANCISCO, CA 94111

This is a communication from the examiner in charge of your application.
COMMISSIONER OF PATENTS AND TRADEMARKS

☒ This application has been examined ☒ Responsive to communication filed on 10/2/92 ☐ This action is made final.

A shortened statutory period for response to this action is set to expire -3- month(s), ——— days from the date of this letter.
Failure to respond within the period for response will cause the application to become abandoned. 35 U.S.C. 133

Part I THE FOLLOWING ATTACHMENT(S) ARE PART OF THIS ACTION:

1. ☒ Notice of References Cited by Examiner, PTO-892.
2. ☐ Notice re Patent Drawing, PTO-948.
3. ☐ Notice of Art Cited by Applicant, PTO-1449.
4. ☐ Notice of Informal Patent Application, Form PTO-152.
5. ☐ Information on How to Effect Drawing Changes, PTO-1474.
6. ☐ _____

Part II SUMMARY OF ACTION

1. ☒ Claims 1, 3, 6-13, 16-30 and 32-70 are pending in the application.
   Of the above, claims 1, 12-13, 16-25, and 48-70 are withdrawn from consideration.
2. ☒ Claims 2, 4, 5, 14, 15 and 31 have been cancelled.
3. ☐ Claims _____ are allowed.
4. ☒ Claims 3, 6-11, 26-30, 32-47 are rejected.
5. ☐ Claims _____ are objected to.
6. ☐ Claims _____ are subject to restriction or election requirement.
7. ☐ This application has been filed with informal drawings under 37 C.F.R. 1.85 which are acceptable for examination purposes.
8. ☐ Formal drawings are required in response to this Office action.
9. ☐ The corrected or substitute drawings have been received on _____. Under 37 C.F.R. 1.84 these drawings are ☐ acceptable. ☐ not acceptable (see explanation or Notice re Patent Drawing, PTO-948).
10. ☐ The proposed additional or substitute sheet(s) of drawings, filed on _____ has (have) been ☐ approved by the examiner. ☐ disapproved by the examiner (see explanation).
11. ☐ The proposed drawing correction, filed on _____, has been ☐ approved. ☐ disapproved (see explanation).
12. ☐ Acknowledgment is made of the claim for priority under U.S.C. 119. The certified copy has ☐ been received ☐ not been received ☐ been filed in parent application, serial no. _____; filed on _____.
13. ☐ Since this application appears to be in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under Ex parte Quayle, 1935 C.D. 11; 453 O.G. 213.
14. ☐ Other

EXAMINER'S ACTION

PTOL-326 (Rev. 9-89)

15. In the communication filed on October 2, 1992, applicants elect Group II with traverse. The claims are properly restricted for the reasons set forth in the last office action.

16. In the communication, applicants stated that claim 26 serves as a linking claim and that a complete examination of claim 26 will require consideration of the art for both groups. The examiner disagrees. In considering restriction, the claims are assumed to be patentable (MPEP 806.05 (a)). The art for Group I and II is separately claimed in claim 1 and 3. In other words, each of the Group I and II does not rely on the other for patentability. In examining claim 26, it does not require to consider the detail claimed in claim 2 which is in Group I. In examining claim 13, it does not require to consider all the details claimed in claim 36. In examining claim 16, it does not require to consider all the details claimed in claim 39. In examining claim 41, it does not require to consider all the details claimed in 21. In examining claim 2, it does not require to consider all the detail claimed in claim 6. In examining claim 24, it does not require to consider all the details claimed in claim 46. In examining claim 3, it does not require to consider the detail in claim 59-62.

17. In conclusion, the independent claims which respectively and solely claim the subject matter in a group is evidence that they do not rely on the detail claimed in the combination claims (one

of the dependent claims in the set of independent claim 26) for patentability. The restriction therefore is proper.

18. The remark in line 11-13 of page 2 of the October 2 communication is not understood. Claim 22 is neither in Group II nor Group X.

19. Claim 12 is inadvertently omitted in the last office action. The error is regretted. Claim 12 should be in Group II.

20. Claims 6, 10, 11, 26-30 and 31-37 are rejected under 35 U.S.C. § 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.

21. With respect to claim 6, the components (means for storing a top item, means for storing a next item and the at least one stack register) of the first push down stack as recited do not appear to render the push down stack to operate as a stack. Note that a stack is such that ~~an~~ item~~s~~ propagate~~s~~ from one end of the stack to another via the stages in the stack. The stack as recited in the claim does not do that. Further, the claim fails to recite how the components of the stack are interconnected so as to form a stack having stages between the input and the output of the stack. The second push down stack has similar defects. Register file is not a stack.

22. Claim 6 further fails to recite how each of the means as recited functionally coacts with each other so as to achieve any

meaningful function or improvement. Although each of the means are recited to be interconnected, no meaningful coact is seen. For example, the means for storing top item of the first stack which is for providing a top item to ALU is recited for providing the same to another stack. The second stack as recited has nothing to do with arithmetic operation. It is not seen why it should receive a top item as the ALU. More example, the second stack is recited to be connected to the means for storing top item bidirectionally. However, the means for storing top item has not been recited for receiving anything from the second stack. It appears to the examiner that they should not be bidirectionally connected and controlled because the means for storing top item is part of another stack and it should receive items from the next stage of its own stack and not from another stack (the second stack).

23. Other claims (claims 27-29 and 37-38, for example) which recite stack have similar defects as claim 6.

24. In claims 10 and 33, it is not clear what is meant by "to provide a microloop in said instruction register". Note that an IR is commonly for storing instruction. Further, it is not seen how the supplying of control/reset signals to counters would provide a microloop in an instruction register.

25. Function of the counter as recited in claims 11 and 34 is not clear. It is not seen how the counter which is recited for

controlling supply of instructions can select variable width operand. Further, claims 11 fails to recite where the variables width operand is stored.

26. In claim 26, function of the multiplexing means is not clear. A multiplexer which is commonly for multiplexing is recited to provide different types of data on a bus. Where do the row addresses, column addresses and data come from and go to?

27. In claim 35, function of the means for fetching is not clear. A fetching means which is commonly for fetching is erroneously recited for assembly and storing instructions.

28. In claim 39, it is not clear what is meant by "different memory access timing for different sizes of DRAM". Is it referring to different storing capacity sizes, to different amount of instructions accessed at a time or to different physical sizes? Further with respect to claim 39, it is not seen how the sensing circuit and the driver circuit as recited can render the microprocessor to provide different sizes of DRAM.

29. Claim 41 is not understood. It is not clear what is meant by "ring counter -- to provide different clock speed -- depending on at least one of temperature, voltage and microprocessor fabrication process --". How does the clock response to the temperature, voltage and microprocessor fabrication process?

30. In claim 42, what is meant by "I/O interface -- to exchange -- signals -- with said I/O interface --"? What is

connected to the I/O interface and exchange with who? Claim 42 further fails to recite how the clock and the I/O interface functionally coact with each other so as to perform any meaningful operation.

31. Claims 44 and 45 fail to recite function of each of the elements recited therein and how they are functionally coact with each other such that desired result can be achieved.

32. Claims 37-38 are rejected under 35 USC 112 and objected to under 37 CFR 1.75 (b) as unduly multiplied.

33. Claims 37-38 are almost identical to parent claim 27-29.

34. The following is a quotation of 35 U.S.C. § 103 which forms the basis for all obviousness rejections set forth in this Office action:

> A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.
>
> Subject matter developed by another person, which qualifies as prior art only under subsection (f) or (g) of section 102 of this title, shall not preclude patentability under this section where the subject matter and the claimed invention were, at the time the invention was made, owned by the same person or subject to an obligation of assignment to the same person.

35. Claims 3, 6-10, 26-30 and 32-33 are rejected under 35 U.S.C. § 103 as being unpatentable over Takahira.

36. See at least Figure 2 and the corresponding description in

the specification of Takahira. The drawing shows a data processing system having a CPU, memory, EEPROM, RAM, ROM, clock circuit, register file, status register, index register X and Y, program counter H and L for fetching instructions, ALU, accumulator, stacks and stack pointer, instruction register, instruction decoder and a bus. With respect to claim 3, Takahira does not specify how many instructions can be fetched per memory cycle.

One of ordinary skill in the art should readily recognize that, for the same machine, more instructions can be fetched if the memory cycle is extended longer. How long a memory cycle should be is merely a matter of design choice because it is dependent on the speed of the elements used and on the engineering design.

37. With respect to claim 7, one of ordinary skill in the art should readily recognize that for the same given amount of time more instructions can be fetched if the previous instruction is not a memory instruction because it is well known that a memory instruction takes longer time to executed.

38. With respect to claim 10, looping is well known in programming art. One of ordinary skill in the art should readily recognize that the processing system of Takahira as shown in Figure 2 is capable of looping because it also has program counters.

39. Claims 11 and 34 are rejected under 35 U.S.C. § 103 as being unpatentable over Takahira in view of Heath.

40. Takahira discloses claim combination set forth above. Takahira does not state whether his operand is of variable length. Heath shows such in lines 31 et seq. of column 5. It would have been obvious to make Takahira's operand variable length because it would be more flexible.

41. Claim 35 and 42 are ~~is~~ rejected under 35 U.S.C. § 103 as being unpatentable over Takahira and Heath in view of Bruinhorst.

42. Takahira and Heath disclose claim combination set forth above. Takahira does not state whether his program in PROM is transferred to RAM. Such is well known in the art as shown by Bruinhorst in lines 43 et seq. of column 15. It would have been obvious to load from PROM to RAM in Takahira as taught by Bruinhorst because it is more flexible in programming.

43. Claims 36, 37 and 38 are rejected under 35 U.S.C. § 103 as being unpatentable over Takahira, Heath, Bruinhorst further in view of Derchak.

44. Takahira, Heath and Bruinhorst disclose claim combination set forth above. Takahira does not show a DMA. DMA is well known in the art. Derchak shows such. It would have been obvious to a person of ordinary skill in the art to incorporate a DMA as taught by Derchak in Takahira because that would render Takahira's system more efficient.

45. Claims 39 and 40 are rejected under 35 U.S.C. § 103 as being unpatentable over Takahira, Heath, Bruinhorst, Derchak further in view of Kimoto.

46. Takahira does not state whether his microprocessor is capable of accessing the memory at a desired variable access time. Such is well known in the art as shown by Kimoto. It would have been obvious to a person of ordinary skill in the art to access the memory of Takahira as taught by Kimoto because the system of Takahira would run more efficiently.

47. Claims 41-45 are rejected under 35 U.S.C. § 103 as being unpatentable over Takahira, Heath, Bruinhorst, Derchak, Kimoto further in view of ~~Kimoto~~ Martin.

48. Takahira does not state whether his clock is of variable clock rate. Variable rate clock is well known in the art. ~~Kimoto~~ Martin shows such. (Fujicom) It would have been obvious to a person of ordinary skill in the art to incorporate a variable speed clock in Takahira's system if the circuits require.

49. With respect to claims 42-43, Takahira shows an I/O interface 13 in Figure 2.

50. Claims 46 and 47 are allowable if the 35 USC 112, second paragraph rejection is overcome.

51. Applicant's arguments with respect to claims 3, 6-11 and 26-30 and 32-45 have been considered but are deemed to be moot in view of the new grounds of rejection.

52. The prior art cited on July 10, 1992 has not been considered because the class and subclass information is missing.

Any inquiry concerning this communication should be directed to David Eng at telephone number (703) 308-1635.

DAVID Y. ENG
PRIMARY EXAMINER
ART UNIT 232

DE/kw
December 29, 1992

TO SEPARATE, HOLD TOP AND BOTTOM EDGES, SNAP--APART AND DISCARD CARBON

FORM PTO-892 (REV. 3-78) U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE	SERIAL NO.	GROUP ART UNIT	ATTACHMENT TO PAPER NUMBER	
NOTICE OF REFERENCES CITED	07/389,334	2315		
	APPLICANT(S) Moore et al			

U.S. PATENT DOCUMENTS

*		DOCUMENT NO.	DATE	NAME	CLASS	SUB-CLASS	FILING DATE IF APPROPRIATE
	A	4067059	1/3/78	Derchak	364	Dig. I.	
	B	4376977	3/15/83	Brunsbort ~~Heath~~	364	Dig. I.	
	C	3603934	9/7/71	Heath	364	Dig I.	
	D	4125871	11/14/78	Martin	364	Dig II.	
	E	4870566	9/26/89	Kimoto	364	Dig I	
	F	5036460	7/30/91	Takahira	~~365~~ 395	425	10/26/88
	G						
	H						
	I						
	J						
	K						

FOREIGN PATENT DOCUMENTS

*		DOCUMENT NO.	DATE	COUNTRY	NAME	CLASS	SUB-CLASS	PERTINENT SHTS. DWG	PERTINENT PP. SPEC.
	L								
	M								
	N								
	O								
	P								
	Q								

OTHER REFERENCES (Including Author, Title, Date, Pertinent Pages, Etc.)

R	
S	
T	
U	

EXAMINER	DATE
BNG	12/92

* A copy of this reference is not being furnished with this office action.
(See Manual of Patent Examining Procedure, section 707.05 (a).)

Exhibit P

101B

PATENT

7/26/93 MC

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of:

CHARLES H. MOORE ET AL.

Serial No. 07/389,334

Filed: August 3, 1989

For: HIGH PERFORMANCE, LOW COST MICROPROCESSOR

Examiner: David Y. Eng

Group Art Unit: 2302

San Francisco, CA

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service as First Class Mail in an envelope addressed to: Commissioner of Patents and Trademarks, Washington, DC 20231 on June 30, 1993.

Signed:

AMENDMENT

Commissioner of Patents
and Trademarks
Washington, D.C. 20231

Sir:

In response to the Office Action dated December 31, 1992, please amend the above application as follows:

In the Claims:

Rewrite claims 3, 6, 7, 10, 11, 26, 27, 29, 30, 33, 34, 35, 36, 39, 41, 42, 44 and 45 as follows:

NANO-001US
Resp. to 3rd. O.A.

Sub C1

3(Twice Amended). A microprocessor system, comprising a central processing unit, a memory, a bus connecting said central processing unit to said memory, and means connected to said bus for fetching instructions for said central processing unit on said bus from said memory, said means for fetching instructions being configured and connected to fetch multiple sequential instructions from said memory in parallel and supply the multiple sequential instructions to said central processing unit during a single memory cycle.

6(Twice Amended). The microprocessor system of Claim 3 in which said central processing unit includes an arithmetic logic unit and a first push down stack connected to said arithmetic logic unit, said first push down stack further including means for storing a top item connected to a first input of said arithmetic logic unit to provide the top item to the first input, means for storing a next item connected to a second input of said arithmetic logic unit to provide the next item to the second input, [and at least one stack register connected to said means for storing a next item to receive the next item from said means for storing a next item when pushed down in said push down stack,] said arithmetic logic unit having an output connected to said means for storing a top item, a second push down stack, said means for storing a top item being connected to provide an input to said second push down stack and a control means connected between said means for storing a top item and said second push down stack for controlling provision of the input to said second push down stack, said second push down stack [comprises] additionally being configured as a register file and said means for storing a top item and said second push down stack additionally configured as the register file are bidirectionally connected.

21(Amended). The microprocessor system of Claim 1 additionally comprising means connected to said means for fetching multiple instructions for determining by decoding the multiple instructions if multiple instructions fetched by said means for fetching multiple instructions require a memory access, said means for fetching multiple instructions fetching additional multiple instructions if decoding the multiple instructions shows that the multiple instructions do not require a memory access.

6 ~~10~~(Twice Amended). The microprocessor system of Claim 5 ~~9~~ additionally comprising a loop counter connected to receive a decrement control signal from said means for decoding, said means for decoding being configured to supply the reset control signal to said counter and the decrement control signal to said loop counter in response to a MICROLOOP instruction in the multiple instructions to provide a microloop within the multiple instructions in said instruction register for a number of repetitions controlled by said loop counter.

7 ~~11~~(Amended). The microprocessor system of Claim 1 ~~3~~ additionally comprising an instruction register for the multiple instructions and a variable width operand to be used with one of the multiple instructions connected to said means for fetching instructions, means connected to said instruction register for supplying the multiple instructions in succession from said instruction register, a counter connected to control said means for supplying the multiple instructions to supply the multiple instructions in succession,

means for decoding the multiple instructions connected to receive the multiple instructions in succession from the means for supplying the multiple instructions, said counter being connected to said means for decoding to receive incrementing and reset control signals from said means for decoding, said means for decoding being configured to control said counter in response to an instruction utilizing [a] the variable width operand stored in said instruction register, and means connected to said counter to select the variable width operand for use with the instruction utilizing the variable width operand in response to said counter.

Sub C2 26 ~~(Twice Amended). A microprocessor system~~, comprising ~~a central~~ processing unit, a dynamic random access memory, a bus connecting said central processing unit to said dynamic random access memory, and multiplexing means on said bus between said central processing unit and said dynamic random access memory, said multiplexing means being connected and configured to provide multiplexed row addresses, column addresses and data on said bus from said central

processing unit to said dynamic random access memory and to provide data from said dynamic random access memory to said central processing unit, and

means connected to said bus for fetching instructions for said central processing unit on said bus from said dynamic random access memory, said means for fetching instructions being configured to fetch multiple sequential instructions from said dynamic random access memory in parallel and supply the multiple instructions to said central processing unit during a single memory cycle.

10
27(Twice Amended). The microprocessor system of Claim 26 in which said central processing unit includes an arithmetic logic unit and a first push down stack connected to said arithmetic logic unit, said first push down stack including means for storing a top item connected to a first input of said arithmetic logic unit to provide the top item to the first input, and means for storing a next item connected to a second input of said arithmetic logic unit to provide the next item to the second input, [and at least one] a remainder of said first push down stack [register] being connected to said means for storing a next item to receive the next item from said means for storing a next item when pushed down in said push down stack, said arithmetic logic unit having an output connected to said means for storing a top item.

11 12 29(Amended). The microprocessor system of Claim 28 10 in which said second push down stack [comprises] is additionally configured as a register file and said means for storing a top item and said second push down stack additionally configured as the register file are bidirectionally connected.

12 13 30(Amended). The microprocessor system of Claim 29 11 12 additionally comprising means connected to said means for fetching multiple instructions for determining by decoding the multiple instructions if multiple instructions fetched by said means for fetching multiple instructions require a memory access, said means for fetching multiple instructions fetching additional multiple instructions if decoding the multiple instructions shows that the multiple instructions do not require a memory access.

14 ~~15~~ 33(Amended). The microprocessor system of Claim ~~13~~ 32 additionally comprising a loop counter connected to receive a decrement control signal from said means for decoding, said means for decoding being configured to supply the reset control signal to said counter and the decrement control signal to said loop counter in response to a MICROLOOP instruction in the multiple instructions within the multiple instructions in said instruction register for a number of repetitions controlled by said loop counter.

34(Twice Amended). The microprocessor system of Claim 33 in which said means for decoding is configured to control said counter in response to [an instruction] one of the multiple instructions utilizing a variable width operand stored in said instruction register with the multiple instructions, said microprocessor system additionally comprising means connected to said counter to select the variable width operand for use with the instruction utilizing the variable width operand in response to a state of said counter resulting from control of said counter by said means for decoding.

35(Amended). The microprocessor system of Claim 34 additionally comprising a programmable read only memory containing instructions connected to said bus, means connected to said bus for fetching instructions for said central processing unit on said bus, said means for fetching instructions including means for assembling a plurality of instructions from said programmable read only memory, [and] storing the plurality of instructions in said dynamic random access memory and subsequently supplying the plurality of instructions from said dynamic random access memory to said central processing unit on said bus.

36(Amended). The microprocessor system of Claim 35 additionally comprising a direct memory access processing unit having the capacity to fetch and execute instructions, said bus connecting said direct memory access processing unit to said dynamic random access memory, said dynamic random access memory containing instructions for said central processing unit and said direct memory access processing unit, said direct memory access processing unit including means

for fetching instructions for said central processing unit on said bus and for fetching instructions for said direct memory access processing unit on said bus.

39(Twice Amended). The microprocessor system of Claim [38] 36 in which said microprocessor system is configured to provide different memory access timing for different storing capacity sizes of said dynamic random access memory by including a sensing circuit and a driver circuit, and an output enable line connected between said dynamic random access memory, said sensing circuit and said driver circuit, said sensing circuit being configured to provide a ready signal when said output enable line reaches a predetermined electrical level after a memory read operation as a function of different capacitance on said bus as a result of the different storing capacity sizes of said dynamic random access memory, said microprocessor system being configured so that said driver circuit provides an enabling signal on said output enable line responsive to the ready signal.

41(Twice Amended). The microprocessor system of Claim 40 in which said microprocessor system is configured to operate at a variable clock speed, said microprocessor system additionally comprising a ring counter variable speed system clock connected to said central processing unit, said central processing unit and said ring counter variable speed system clock being provided in a single integrated circuit, said ring counter variable speed system clock being configured to provide different clock speed to said central processing unit as a result of transistor propagation delays, depending on at least one of temperature of said single integrated circuit, voltage and microprocessor fabrication process for said single integrated circuit.

42(Amended). The microprocessor system of Claim ~~41~~ 23 additionally comprising an input/output interface connected between said microprocessor system and an external memory bus to exchange coupling control signals, addresses and data [with] between said central processing unit and said input/output interface, and a second clock independent of said ring counter variable speed system clock

connected to said input/output interface to provide clock signals for operation of said input/output interface asynchronously from said central processing unit.

44(Twice Amended). The microprocessor system of Claim 43 in which said first push down stack has a first plurality of stack registers having stack memory elements configured as latches, a second plurality of stack registers having stack memory elements configured as a random access memory, said first and second plurality of stack registers and said central processing unit being provided in a single integrated circuit with a top one of said second plurality of stack registers being connected to said a bottom one of said first plurality of stack registers, and a third plurality of stack registers having stack memory elements configured as a random access memory external to said single integrated circuit, with a top one of said third plurality of stack registers being connected to a bottom one of said second plurality of stack registers, said microprocessor system being configured to operate said first, second and third plurality of stack registers hierarchically as interconnected stacks.

45(Twice Amended). The microprocessor system of Claim 44 additionally comprising a first pointer connected to said first plurality of stack registers, a second pointer connected to said second plurality of stack registers, and a third pointer connected to said third plurality of stack registers, said microprocessor system being configured to operate said first, second and third plurality of stack registers hierarchically as interconnected stacks by having said central processing unit being connected to pop items from said first plurality of stack registers, said first stack pointer being connected to said second stack pointer to pop a first plurality of items from said second plurality of stack registers when said first plurality of stack registers are empty from successive pop operations by said central processing unit, said second stack pointer being connected to said third stack pointer to pop a second plurality of items from said third plurality of stack registers when said second plurality of stack registers are empty from successive pop operations by said central processing unit.

Cancel claims 37-38.

Add the following new claims:

~~3~~ 2 ~~71~~. The microprocessor system of Claim ~~71~~ in which the decoding determines if the multiple instructions do not require a memory access by a state of a bit of each of the multiple instructions.

~~72~~ 4. The microprocessor system of Claim ~~71~~ 3 in which the bit is a most significant bit of the multiple instructions.

~~73~~ 16 ~~17~~. The microprocessor system of Claim ~~30~~ 12 ~~13~~ in which the decoding determines if the multiple instructions do not require a memory access by a state of a bit of each of the multiple instructions.

~~74~~ 17 ~~18~~. The microprocessor system of Claim ~~73~~ ~~17~~ 16 in which the bit is a most significant bit of the multiple instructions.

~~75~~ 20 ~~21~~. The microprocessor system of Claim ~~36~~ 19 ~~20~~ additionally comprising a variable speed system clock connected to said central processing unit and a fixed speed system clock connected to control said means for fetching instructions for said central processing unit and for fetching instructions for said direct memory access processing unit.--.

B13

REMARKS

Claims 3, 6-11, 26-30 and 32-47 are presently under examination in the application. The allowability of claims 46 and 47, if amended to overcome the rejection under 35 U.S.C. § 112, is noted. Claims 37-38 have been canceled to advance the prosecution of the application.

Claims 6, 10, 11, 26-30 and 31-37 were rejected under 35 U.S.C. § 112 as indefinite. In response, claims 6, 10, 11, 26, 27, 29, 34, 35, 39, 41, 42, 44 and 45 have been rewritten to define the invention with more particularity.

In claim 6, the first push down stack is now recited as further including the means for storing a top item connected to a first input of the arithmetic logic unit to provide the top item to the first input and the means for storing a next item connected to a second input of said arithmetic logic unit to provide the next item to the second input. Thus, as the Examiner correctly notes, these items do not render the first push down stack to operate as a stack. These items are in addition to the conventional construction of the first push down stack which allow it to operate as a stack. Similarly, the second push down stack is now recited as additionally being configured as a register file. In both cases, the recited language is in addition to conventional organization of the stacks which allow them to operate as stacks.

The Examiner is correct that a register file is not a stack. However, a stack which posses an organization which emulates registers is still a stack. Such an organization conveys the benefits of both stacks and registers while avoiding the limitations of either. Since the elements of both stacks that allow them to operate as stacks are conventional, they have not been recited beyond specifying these elements as being push down stacks.

The recited structure of claim 6 permits use of the technique of placing local variables on the stack, which allows automatic nesting of procedures and their local variables, simply by pushing new variables on the stack to allocate new space.

The two stacks as now claimed serve distinct functions. The first push down stack is exemplified by the stack 74 in Figures 2 and 13. The stack 74 in fact allows arithmetic operations to be carried out on operands supplied from it to the ALU and receives ALU results as a result of the recited connections.

The second push down stack is exemplified by the stack 134 in Figures 2 and 13. The RSTACK 134 stores return addresses for subroutine nesting as well as local storage for subroutines.

The defined relationship between the two stacks is that they are linked to a bidirectional buffer, which allows the stacks to exchange individual contents. Two instructions, POP-STACK-PUSH-RSTACK and POP-RSTACK-PUSH-STACK,

move the top items of one stack to the top item of the other. WRITE-LOCAL-VARIABLE and READ-LOCAL-VARIABLE write or read the top arithmetic stack item to the local variables stored on the RSTACK 134.

The underlying rationale for using two stacks is to remove the processing bottleneck found in single stack machines. This much is taught, for example, by the Moore et al prior art U.S. Patent 5,070,451, discussed in the Information Disclosure Statement of record. Those familiar with languages like FORTH which make extensive use of the dual stack concept are comfortable with the dual stack arrangement. The addition of a register array embedded in the second stack as claimed eliminates the stack machine problem of either storing local variables on the stack or in off-chip memory. Storing local variables on either stack becomes very clumsy with prior art dual stacks once the number of variables exceeds three. Off-chip variables access far slower than on-chip registers.

Similarly, claims 27-29 have been rewritten to specify that the first push down stack functions both to supply operands to and receive results from the ALU, as well as being a conventional push down stack. The second push down stack is also now specified as operating both as a register file and as a conventional push down stack.

Claims 10 and 33 have been rewritten to require that the MICROLOOP instruction in the multiple instructions provide a microloop within the multiple instructions in the instruction register and that the loop counter controls the number of repetitions of the microloop. It is believed that the provision of the microloop and the function of the loop counter are now clear.

Claims 11 and 34 have been rewritten to specify that the variable width operand is stored in the instruction register and that the variable width operand is selected for use with the instruction utilizing the variable width operand in response to the counter. As is explained at page 33 of the specification with reference to Figure 20, the instruction decoder tests the counter to determine the position of a JUMP op-code in the four instruction group and assumes that the remaining bits in the instruction group are the JUMP operand. Since the JUMP instruction's position in the four instruction group determines the length of the operand, a single JUMP op-

code may have three different length operands. With these amendments, the function of the counter in these claims is now believed to be clear.

Claim 26 has been rewritten to specify that the multiplexer provides multiplexed row addresses, column addresses and data on the bus from the central processing unit to the dynamic random access memory and data from the dynamic random access memory to the central processing unit. The instructions are now specified as being from the dynamic random access memory. The function of the multiplexer is now believed to be clear, and the locations from which the row addresses, column addresses, data and instructions come are now specified.

Claim 35 has been rewritten to specify that the means for fetching instructions includes means for assembling a plurality of instructions from the programmable read only memory, storing the plurality of instructions in the dynamic random access memory and subsequently supplying the plurality of instructions from the dynamic random access memory to the central processing unit on said bus. Thus, it is now clear that the means for fetching instructions is capable of fetching a plurality of instructions because it also includes a means for assembling the plurality of instructions from the programmable read only memory and storing the plurality of instructions in the random access memory, from which they are supplied to the central processing unit. The function of the means for fetching is now believed to be clear.

Claim 39 now specifies different storing capacity sizes of the dynamic random access memory and that the ready signal is provided when the output enable line reaches a predetermined electrical level after a memory read operation as a function of different capacitance on the bus as a result of the different storing capacity sizes of the dynamic random access memory. The different sizes of DRAM and the ability of the system to provide different memory timing for the different sizes of DRAM is now believed to be clear.

Claim 41 now specifies that the ring counter variable speed system clock is configured to provide different clock speed to the central processing unit as a result of transistor propagation delays, depending on at least one of temperature, voltage and microprocessor fabrication process for the single integrated circuit. The clock thus indirectly responds to temperature, voltage and microprocessor fabrication

process by responding to transistor propagation delays, which are determined by those parameters. Claim 41 is therefore believed to be clarified.

Claim 42 now recites an input/output interface connected between the microprocessor system and an external memory bus to exchange coupling control signals, addresses and data between the central processing unit and the input/output interface. The claim now further calls for a second clock independent of the ring counter variable speed system clock connected to the input/output interface to provide clock signals for operation of the input/output interface asynchronously from the central processing unit. It is therefore believed that the function of the input/output interface is now clear and the function of the second clock to allow asynchronous operation of the input/output interface with respect to the central processing unit has been clarified.

Claims 44 and 45 have been rewritten to specify the interconnections among the first, second and third plurality of stack registers and to specify that the microprocessor system is configured to operate the first, second and third plurality of stack registers hierarchically as interconnected stacks. It is believed that these claims now recite the function of the elements and how they functionally coact with one another to achieve a desired result.

Based on the above changes to the claims and remarks, it is believed that all of the claims are now definite in form. The rejection of claims 6, 10, 11, 26-30 and 31-37 under 35 U.S.C. § 112 is believed to be overcome.

Claims 3, 6-10, 26-30 and 32-33 were rejected under 35 U.S.C. § 103 as unpatentable over Takahira et al., U.S. Patent 5,036,460. In response, in addition to the above-discussed changes to the claims, claims 3, 7, 26 and 30 have been rewritten to define the invention better over the prior art, and new claims 71-74 have been added to provide more complete protection for the invention. This rejection is believed to be overcome by the above changes to the claims and the following remarks.

Claim 3 has been rewritten to specify that the means for fetching is configured and connected to fetch multiple sequential instructions from the memory in parallel and to supply the multiple sequential instructions to the central processing unit during a single memory cycle. A system including such a means for fetching is

not taught or suggested by Takahira et al. Takahira et al. disclose only the parallel movement of data within a memory, solely for the purpose of writing to an EEPROM. The system of claim 3 is a low-cost technique which allows balancing a very fast CPU with a very slow memory to produce a fast computing system.

In the rejection, the Examiner argues that it would be obvious to extend the memory cycle in Takahira et al. to allow multiple instruction fetching. This argument misunderstands the subject matter of the claim. As claimed, the multiple instructions are fetched in parallel during a single memory cycle. Because they are fetched in parallel, no substantial extension of the memory cycle is required to fetch the multiple instructions. Even if the Examiner is correct that it would be obvious to extend a memory cycle to allow multiple instruction fetching, doing so as posited by the Examiner would not give the claimed subject matter.

The rejection contains no separate discussion of the subject matter of claim 6. The above clarifying changes made to that claim and remarks with respect to the § 112 rejection also make it clear that the subject matter of claim 6 is not suggested by Takahira et al. That reference contains no teaching or suggestion of dual stacks which are operable as both stacks and registers.

Claims 7 and 30 have been rewritten to require determining by decoding the multiple instructions if multiple instructions fetched by the means for fetching multiple instructions require a memory access. Making the determination in this manner means that it can be done in 2.5 nanoseconds in the described embodiment, as pointed out at page 38, line 19 of the specification. No such determination, whether or not done by decoding, is taught or suggested by Takahira et al. New claims 71-74 provide further details on how the decoding is carried out, and are also not taught or suggested by Takahira et al.

Claim 9 adds structure to the microprocessor system of claim 3 to handle SKIP instructions. Claim 32 adds the same structure to the microprocessor system of claim 30. Takahira et al. contains no teaching of how SKIP instructions are handled in the system there disclosed.

Claim 10 adds to the microprocessor system of claim 9 a loop counter for controlling the number of repetitions of a microloop within the multiple instructions. In the rejection of this claim, the Examiner equates the loop counter to the program

counter in Takahira et al, used for controlling software looping. In fact, claim 10 is directed to a hardware accelerator for microloop repetition and is not suggested by a conventional program counter to control software looping.

Claim 26 includes the fetching of multiple sequential instructions from the memory in parallel and supplying the multiple sequential instructions to the central processing unit during a single memory cycle as in claim 3 and the provision of a multiplexed bus connected between the memory and the central processing unit for supplying addresses and data between the central processing unit and the memory. No such combination is shown or suggested by Takahira. The above comments with respect to the rejection of claim 6 are equally applicable to the subject matter added to the microprocessor system by claim 29. The above comments with respect to the rejection of claim 10 are equally applicable to the subject matter added to the microprocessor system by claim 33.

Based on the above changes to the claims and remarks, the rejection of claims 3, 6-10, 26-30 and 32-33 under 35 U.S.C. § 103 as unpatentable over Takahira et al. is believed to be overcome.

Claims 11 and 34 were rejected under 35 U.S.C. § 103 as unpatentable over Takahira et al. in view of Heath, U.S. Patent 3,603,934. The above remarks with respect to the rejection of claim 3 are equally applicable to the rejection of claim 11. The above amendments to these claims clarify the function of the counter in these claims to control selection of the variable width operand by determining the position of the instruction utilizing the variable width operand. The bare mention of variable width operands in Heath fails to teach or suggest the subject matter added to the microprocessor system by claims 11 and 34. This rejection is believed to be overcome.

Claims 12 and 35 were rejected under 35 U.S.C. § 103 as unpatentable over Takahira et al. and Heath in view of Bruinshorst, U.S. Patent 4,376,977. The loading of instructions from a PROM to RAM as disclosed by Bruinshorst fails to teach or suggest the use of a bus in unmultiplexed form for reading instructions from a PROM and the dynamic reconfiguration of the same bus to multiplexed form for row addresses, column addresses and data during the transmission of the instructions to the RAM, as recited in claim 12. The above remarks with respect to

the rejections of claims 34, 33, 32, 30, 29, 28, 27 and 26 are equally applicable to the rejection of claim 35. The rejection of claims 12 and 35 is therefore believed to be overcome.

Claims 36, 37 and 38 were rejected under 35 U.S.C. § 103 as unpatentable over Takahira et al., Heath, Bruinshorst, further in view of Derchak, U.S. Patent 4,067,059. In response, claim 36 has been rewritten to distinguish the invention better over the prior art. Claims 37-38 have been canceled to advance the prosecution of the application. New claim 75 has been added to provide more complete protection for the invention. The above comments with respect to the rejections of claims 35, 34, 33, 32, 30, 29, 28, 27 and 26 are equally applicable to this rejection.

As described at page 15, lines 8-11, claim 36 has been rewritten to specify that the direct memory access processing unit is capable of fetching and executing instructions. Like the central processing unit, it is therefore also a stored program processing unit. In making the rejection, the Examiner has equated the subject matter added to the system by claim 36 to direct memory access, as shown by Derchak. However, as pointed out by Derchak at column 1, lines 29-39, conventional direct memory access controllers do not have the capability to fetch their own instructions, as required by rewritten claim 36. Instead, instructions and data necessary for the direct memory access controller are supplied to the direct memory access controller by the central processing unit. The Derchak patent deals with the sharing of a direct memory access controller among multiple peripherals, and there is no indication that the direct memory access controllers in Derchak have the capability to fetch their own instructions, as claimed.

New claim 75 further distinguishes from these references by specifying a variable speed system clock for the central processing unit and a fixed speed system clock connected to control the means for fetching instructions for said central processing unit and for fetching instructions for said direct memory access processing unit, as shown in Figure 17.

The system of claim 75 allows a division of computing work between that which is real-time (I/O) and that which is not (everything else). Real-time

computing work is driven either by time or external events. The crystal clock in Figure 17 is the time base for the I/O interface 432.

Everything else, such as calculating, sorting and performing logical operations, is by nature asyncronous with the real world. The optimal implementation for such computing work is whatever produces results faster. The variable speed clock for the CPU 70 in Figure 17 clocks execution as fast as possible, given the voltage, temperature and process parameters of the CPU, without having to be concerned with slowing down for I/O considerations.

The invention of claim 75 achieves efficiencies by dividing the real-time component of the direct memory access processing unit from the non-real time component of the central processing unit. In this system, two processors with independent instruction streams, independent program counters and even independent instruction sets coexist in a loosely coupled fashion, synchronizing only when necessary to exchange information.

If the CPU clock were the same as the direct memory access clock, CPU instructions would have to be slowed down to less than their fastest speed, because, while the operation of a crystal is near constant over voltage and temperature, the operation of transistors is not. If the direct memory access clock were the same as the variable speed CPU clock, no time precise direct memory access could be performed because no time standard would exist.

For these reasons, an independent basis for patentability of new claim 75 over this prior art is present. Based on the above changes to claim 36 and remarks, its rejection under 35 U.S.C. § 103 is believed to be overcome.

Claims 39 and 40 were rejected under 35 U.S.C. § 103 as unpatentable over Takahira et al., Heath, Bruinshorst, Derchak, further in view of Kimoto et al., U.S. Patent 4,870,562. The above comments with respect to the rejection of claim 36 are equally applicable to this rejection. Claim 39 has been rewritten to make it clear that the different memory access timing is provided for different storing capacity sizes of the dynamic random access memory as a function of different capacitance on the bus as a result of the different storing capacity sizes of the dynamic random access memory. No such operation is suggested by Kimoto et al. Kimoto et al. involves executing instructions at faster speeds when fetched from on-

board memory and slower speeds when fetched from external memory. This is essentially an instruction cache patent and has nothing to do with sensing memory expansion by measuring capacitance attached to a memory bus, as claimed. The rejection of claims 39 and 40 is believed to be overcome.

Claims 41-45 were rejected under 35 U.S.C. § 103 as unpatentable over Takahira et al., Heath, Bruinshorst, Derchak, Kimoto et al., further in view of Martin. The above remarks with respect to the rejection of claims 39 and 40 are equally applicable to this rejection. Additionally, claim 41 has been rewritten to require that the ring counter variable speed system clock provide a different clock speed to the central processing unit as a result of transistor propagation delays depending on temperature of the integrated circuit containing the microprocessor system.

The Martin patent is directed to a similar problem as claims 41-45, but an external temperature sensor measures ambient temperature, which is at best only an indirect approximation of the integrated circuit temperature. In claim 41, the ring counter in the integrated circuit serves as a direct measure of propagation delays, which are a function of the integrated circuit temperature. No such direct measurement of integrated circuit temperature is contemplated by Martin, nor is varying clock speed on the basis of voltage or integrated circuit fabrication process, the other two factors recited in claim 41. The above comments with respect to the rejection of claim 36 and the subject matter of new claim 75 are applicable to the rejection of claims 42-43. None of these references teach or suggest the hierarchically interconnected stack registers of claims 44-45. The rejection of claims 41-45 is believed to be overcome.

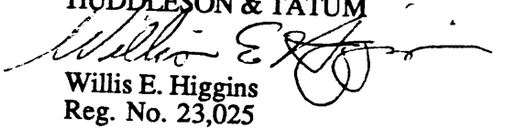
In the Office Action, the Examiner indicates that the references cited in the Information Disclosure Statement have not been considered because the class and subclass information for the references was not supplied. In response, PTO Form 1449 is being resubmitted to include this information. Applicants note that copies of the references were included with the Information Disclosure Statement, and it is therefore not clear why the class and subclass information was necessary to consider these references.

A three-month extension of time to reply to the Office Action is requested. A check for $420.00 to cover the fee for this extension is enclosed. Please charge any additional extension of time fee or credit any overpayment to Deposit Account No. 03-3117(Order No. NANO-001US). A copy of this page is enclosed for charging purposes.

All of the claims in the application are believed to be patentable over the prior art. This application is believed to be in condition for allowance, and allowance is solicited.

Respectfully submitted,

COOLEY GODWARD CASTRO
HUDDLESON & TATUM



Willis E. Higgins
Reg. No. 23,025

Five Palo Alto Square
Fourth Floor
Palo Alto, California 94306-2155
Telephone: (415) 843-5145