THE ICS MULTUM COMPUTER

(to the best of my recollection)

SOME HISTORICAL AND TECHNICAL BACKGROUND

When ICT merged with English Electric Computers in 1968, EEC's real–time department was hived off to another newly–synthesised company, Marconi Elliott Computer Systems. A group of EEC engineers who preferred not to relocate from Kidsgrove to Borehamwood decided that—rather than work for MECS—they would start a new company: Information Computer Systems (ICS). ICS took over disused railway works in Crewe and proceeded to develop the Multum, a series of 16–bit real–time minicomputers that ranged from the basic Arithmetic Logic Processor/1 (ALP/1), comparable with a PDP–11/20, to the ALP/3, an 8MHz CPU with virtual memory and floating point hardware.

Designed as a multiprocessor complex, the Multum was more powerful than any contemporary PDP–11. In a maximal configuration it would have had 4 store modules of 128K bytes each, and 8 processors, connected by a crossbar switch that allowed all 4 stores to be accessed simultaneously. Of the 8 processors, up to 4 could be CPUs in any mix of ALP/1s, ALP/2s and ALP/3s (the ALP/2 was an ALP/3 without the floating point feature). The remaining processors could be any mix of Communications Processors, supporting asynchronous and synchronous line units; and Multiplexed I/O Processors, which were similar to IBM's Multiplexor Channels.

The Multum had some generic similarities, and some common roots, with the GEC 4080. An open question is how it related to another contemporary, the Marconi Locus 16, which also used the distinctive 'ALP' term for a CPU. Unlike the 4080, the Multum did not have a microcoded operating system kernel. That functionality was implemented in software, with the help of a very capable context switching instruction and a memory map providing each process with 16 variable length areas of up to 4K words. Each area could have its own access status, could be independently sized, and could be independently located in RAM, which was initially implemented as core storage with a 650ns cycle time.

The Multum was announced in early 1972, and by early 1973 the Computing Science Department at Glasgow University had acquired a pre–production model, with a view to conducting research in operating systems and compilers. So far as I know, this was the only ALP/3 configuration ever built. It had:

- an ALP/3 processor,
- one 64K word (128K byte) RAM module,
- a CDC video terminal as the control console,
- a CDC 300 lines/minute line printer,
- a Documation 300 cards/minute card reader,
- an Elliott 1000 characters/second paper tape reader,
- a Facit 110 characters/second paper tape punch, and
- a CDC 60M byte disk drive.

There was no crossbar switch, Communications Processor or Multiplexed I/O Processor. Instead, all the individual device controllers were fitted to the CPU's Basic I/O Processor, a DMA channel implemented in the CPU microcode.

The Computing Science Department contracted with ICS to develop a general purpose operating system to complement their real–time system. As a first step, a Multum Pascal compiler was half–bootstrapped (by Robert Cupples, John Cavouras and myself) from the ICL 1900 Series Pascal implementation by Jim Welsh and Colm Quinn [1]. This made Multum Pascal the grandchild of Wirth's famous CDC implementation [2]; it was probably the world's third Pascal compiler, and was almost certainly the first on a 16–bit minicomputer [3].

The operating system project got as far as testing a microkernel–based Executive written in Symbolic Usercode Language, the Multum assembler; specifying major OS components; and prototyping a filing system written in Pascal.

Iain Smith's microkernel implementation was a piece of virtuoso programming. It had some syntax errors on its first assembly; it assembled, but quickly failed on a second attempt; and it ran perfectly at a third attempt, supporting a system of about a dozen concurrently executing processes. These consisted of a device driver process for each of the I/O devices, working in privileged state; and a set of processes working in user state, each of which copied data from one of the input devices to one of the output devices.

The Basic I/O Processor of the ALP/3 was a selector channel (in IBM terms). That is, it was dedicated to one DMA transfer at a time. To gain the efficiency offered by keeping I/O devices in simultaneous operation, the BIOP was timeshared by the microkernel, in a manner closely analogous to its timesharing of the CPU. This created a virtual BIOP for each peripheral. When the system test was running there were no discontinuities in the smooth operation of the various I/O devices, which performed close to their rated speeds. Virtualizing the BIOP was made feasible by the fact that each controller had adequate buffering to keep its device going while it was not being serviced by the BIOP.

Then calamity struck.

Testing the microkernel revealed a bug in the DMA channel, which shared a working register with the ALP's address generation microcode but failed to access it under mutual exclusion. ICS sent an engineer to make the necessary change, but struggled to implement it on the hand–soldered wiring of the pre–production hardware.

Next a banking crisis blew up in the USA, and in August 1973 the financial backers of ICS abruptly withdrew their funding. Overnight the company was rendered insolvent and ceased trading. *Plus ça change …*

The microcode bug never was fixed, the computer rapidly deteriorated, and the project drew to a dispiriting close.

REGISTER STRUCTURE

The Multum ALP/3 has the following working–register set:

`A` The primary 16–bit arithmetic register

`B` The primary 16–bit index register

`E/F` The 32–bit extended arithmetic register obtained by concatenating `A` and `B` (notated `F` for floating point orders)

`X` A secondary 16–bit index register

`Y` A secondary 16–bit index register

`S` The 16–bit program counter (**s**equence) register

`P` The 16–bit **p**rocedure stack frame base register

`C` The 16–bit **c**onditions register that contains CPU state bits such as Carry, Arithmetic Overflow, and so on.

There is only one copy of these registers, so they must be saved and restored as part of a context switch.

VIRTUAL ADDRESSING

Each **program level** (process) works in its own 16–bit address space. The addressable unit is the 16–bit word, so that the virtual address space encompasses 64K words. It is divided into 16 virtual **domains** of 4K words, each of which is independently relocatable within physical storage, maps a physical area varying in increments of 64 words, and can be set to expand towards lower or towards higher addresses. Each domain has its own access status, which is one of No Access (NA); Read Only (RO) for constant data, neither executable nor writable; Code Only (CO) for executable code, but allowing embedded constants to be read; and Read Write (RW), but not executable. There are 16 domain–descriptor registers, which are loaded, as a group, by the privileged `ENTL` instruction.

Domain 1 (the range of virtual addresses from 4K to 8K–1) contains a level's **master segment**. This is its 'process control block': a store for the most recently saved contents of the level's working registers and its domain descriptors, it normally has RO status to the level. On a context switch from level *p* to level *q*, the Executive saves the working registers in the master segment of *p*, then defines a master segment for *q* by loading domain register 1 for *q*, and finally restores the contents of the working registers and (other) domain registers for *q* from its master segment, using `ENTL`.

In my view, making the process control block a segment, and incorporating it into the virtual address space of the process, was the most serious misjudgement in the Multum architecture. A directly accessible master segment has little value to a process, so a substantial fraction of its virtual address space is made useless; and that error is compounded by siting it in the interior of the address space, rather than at one end or the other, thereby reducing the size of the largest contiguous structure that can be supported from 64K to 56K.

Because of the master segment in domain 1, domain 0 is an island cut off from the mainland of domains 2–15. The ICS Executive used domain 0 as a scratch mailbox into which processes received **messages**, each a few words of data sent by another process. The microkernel Executive that Iain Smith and I developed took a different line, rejecting this profligate use of scarce virtual addresses. We dedicated domain 0 to a shared library of application support routines, including routines that provided Executive API shims for programs compiled from Pascal and other high–level languages. A single copy of these re–entrant routines was mapped into domain 0 of every process, thus economising physical storage as well as virtual addresses. Short messages could instead be copied into any writable area. Both Executives allowed a message to optionally attach a segment, and a process receiving such a message could map the segment into its own virtual address space, providing for very efficient communication of bulk data.

ADDRESS GENERATION AND ADDRESSING MODES

A 16-bit Multum instruction has room for, at most, 8 bits of address constant; so almost all addresses have to be derived from the contents of registers. Registers `A`, `B`, `P`, `S`, `X` and `Y` can be used in this way, but they are few and have other duties that prevent their customary employment as bases. In mitigation, the Multum has the concept of **memory–held registers**. The `P` register contains an address which is conventionally the base for the local variables of a procedure. The 8 locations at `[P]+0` through `[P]+7` are that procedure's memory–held registers, and addressing modes are provided to allow them to be used as addressing bases. By initializing these words with suitable values, every procedure has at its disposal 8 independent base pointers, the trade off being an additional store cycle to fetch the pointer value. (The 8 locations at `[P]+8` through `[P]+F` are also known as memory–held registers, but have much more limited functionality and cannot be used as address bases.)

A Multum virtual address identifies a 16–bit word, not a byte. An un–indexed byte handling order always accesses the most significant half of the word in store. However, if the addressing mode specifies an index register, the content of the latter is taken to be a byte offset from the base address. This is achieved by shifting the index value right one place before adding it to the base address, to locate the word that contains the byte. The bit that is shifted off at the right selects which of the two bytes in that word is to be the operand, byte 1 being the less significant halfword. Similarly, some orders access double–word operands, and here the address scaling works in the opposite manner: the index is taken to be a double–word offset, and so it is shifted left one place before being added to the base address. No shifting of the index value is done for word operands. So the effect of an order with base address ***B*** and index ***i*** is always to access the ***i***–th item of the implied operand size, with the zeroth item being located at the virtual address ***B***.

NOTATIONAL CONVENTIONS

All instructions occupy 16 bits, numbered 0 through F in the machine code diagrams below. The bits comprising a field of a machine code order are flagged with the same italic letter, e.g. the eight *d*s distributed across the *displacement* field in the first format below. In the description of the semantics of the order, a bold italic letter, e.g. ***d***, denotes the value of the corresponding field as a whole.

[*x*] denotes the contents of *x*, whether it be a register or a store location.

ORDERS WITH A LITERAL OPERAND OR A STORED OPERAND

For these orders, *mnem* represents one of the following operations:

f	*mnem*	OPERATION
01	`STBS`	store `B` in a word
02	`STHS`	store lower halfword of `A` in a byte
03	`STES`	store `E` in a double–word
04	`JUMP`	load `S` with effective address
05	`LODP`	load `P` with effective address
06	`STAS`	store `A` in a word
07	`LINK`	store `S+1` in a word, to act as a subroutine return link
08	`SETE`	load double–word operand to `E`
09	`ADDE`	add double–word operand to `E`
0A	`SUBE`	subtract double–word operand from `E`
0B	`SETB`	load word operand to `B`
0C	`ADDF`	add floating double–word operand to `E`
0D	`SUBF`	subtract floating double–word operand from `E`
0E	`MLTF`	multiply floating double–word operand into `E`
0F	`DIVF`	divide floating double–word operand into `E`
10	*reserved for expansion*	
11	`SETH`	load byte operand to lower halfword of `A`
12	`CASH`	compare byte operand with lower halfword of `A` and skip {`S` := [`S`] + 1} if equal
13	`SETA`	load word operand to `A`
14	`ADDA`	add word operand to `A`
15	`SUBA`	subtract word operand from `A`
16	`MLTA`	multiply word operand into `A`, giving a 32–bit product in `E`
17	`DIVE`	divide word operand into `E`, giving a quotient in `A` and remainder in `B`
18	`ANDA`	logical **and** word operand to `A`
19	`LORA`	logical **or** word operand to `A`
1A	`NEVA`	logical **exclusive or** word operand to `A` (**n**ot **e**qui**v**alent)
1B	`MASA`	mask word operand with `A` and skip {`S` := [`S`] + 1} if 0
1C	`CASA`	compare word operand with `A` and skip {`S` := [`S`] + 1} if equal
1D	`EXAS`	exchange `A` with word operand
1E	`INCS`	increment word operand and skip {`S` := [`S`] + 1} if 0
1F	`DECS`	decrement word operand and skip {`S` := [`S`] + 1} if 0

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
f	*f*	*f*	*f*	*f*	1	*m*	*m*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*

For store–addressing modes, the effective address, ***e***, is computed as [***r***] + ***d***, –128 ≤ ***d*** ≤ 127, where ***r*** is the register supplying the base address; or as [[***r***] + ***d***] for the indirect modes.

USERCODE FORMAT (if ***d*** is an identifier or assembly–time expression, it must be enclosed in parentheses)

mnem `L`***d*** {for ***m***=0, fetch–type orders, the literal operand is 0 ≤ ***d*** ≤ 255}

mnem `S`***d***`I` {for ***m***=0, store–type orders, ***e***=[[`S`] + ***d***]}

mnem `S`***d*** {for ***m***=1, ***e***=[`S`] + ***d***}

mnem `P`***d*** {for ***m***=2, ***e***=[`P`] + ***d***}

mnem `P`***d***`I` {for ***m***=3, ***e***=[[`P`] + ***d***]}

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
f	*f*	*f*	*f*	*f*	0	1	*i*	*i*	*i*	*d*	*d*	*d*	*d*	*d*	*d*

For this mode the effective address, ***e***, is given by [M***i***] + ***d***, ***d*** ≤ 63, where M***i*** is the ***i***–th memory–held register, *i.e.* [M***i***] is [[P] + ***i***].

USERCODE FORMAT

mnem M***i*** ***d*** { ***e***=[M***i***] + ***d***}

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
f	*f*	*f*	*f*	*f*	0	0	*i*	*i*	*i*	*m*	*m*	*m*	*j*	*j*	*j*

For these modes the base address is given by the sum of two memory–held registers [M***i***] + [M***j***], which may be used directly, or as an indirect address, and then optionally indexed by the contents of the X register or the Y register, which is scaled appropriately (denoted by [X]*, for example).

USERCODE FORMAT

mnem M***i*** M***j*** {for ***m***=0, ***e***=[M***i***] + [M***j***]}
mnem M***i*** M***j***I {for ***m***=1, ***e***=[[M***i***] + [M***j***]]}
mnem M***i*** M***j***X {for ***m***=2, ***e***=[M***i***] + [M***j***] + [X]*}
mnem M***i*** M***j***IX {for ***m***=3, ***e***=[[M***i***] + [M***j***]] + [X]*}
mnem M***i*** M***j***Y {for ***m***=4, ***e***=[M***i***] + [M***j***] + [Y]*}
mnem M***i*** M***j***IY {for ***m***=5, ***e***=[[M***i***] + [M***j***]] + [Y]*}

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
f	*f*	*f*	*f*	*f*	0	0	*i*	*i*	*i*	1	1	0	*m*	*j*	*j*

For these modes the effective address is the sum of a memory–held register and a working register; *i.e.*, [M***i***] + [***r***]*; where ***r*** may be: ***j***=0 → A, 1 → B, 2 → X, 3 → Y, is scaled appropriately, and is optionally auto–incremented.

USERCODE FORMAT

mnem M***ir*** {for ***m***=0, ***e***=[M***i***] + [***r***]*}
mnem M***ir*** **+** {for ***m***=1, ***e***=[M***i***] + [***r***]*; ***r*** := [***r***] + 1}

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
f	*f*	*f*	*f*	*f*	0	0	*i*	*i*	*i*	1	1	1	*m*	*j*	*j*

For these modes the effective address is the sum of two working registers; *i.e.*, [***s***] + [***r***]*, where:
the base, ***s*** may be: ***i***=0 → A, 1 → B, 2 → X, 3 → Y, 4 → S, 5 → P, 6 → Z (= 0), 7 → D (= –1); and
the index, ***r*** may be: ***j***=0 → A, 1 → B, 2 → X, 3 → Y, is scaled appropriately, and is optionally auto–incremented.

USERCODE FORMAT

mnem ***sr*** {for ***m***=0, ***e***=[***s***] + [***r***]*}
mnem ***sr*** **+** {for ***m***=1, ***e***=[***s***] + [***r***]*; ***r*** := [***r***] + 1}

LITERAL OR MEMORY–HELD REGISTER BIT–NUMBER PARAMETER: RE/SET OR TEST BIT

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	1	*f*	*f*	*i*	*i*	*m*	*j*	*j*	*j*	*j*

For these orders, the parameter ***p*** is either the literal ***j*** or [M***j***].
r is: ***i***=0 → A; 1 → B; 2 → X; 3 → Y.

USERCODE FORMAT

mnem L***j*** {for ***m***=0}
mnem M***j*** {for ***m***=1}

INSTRUCTIONS

f	*mnem*	OPERATION
0	BSO*r*	Set bit ***p*** of ***r*** to 1
1	BSZ*r*	Reset bit ***p*** of ***r*** to 0
2	SOB*r*	Test bit ***p*** of ***r*** and skip {S := [S] + 1} if 1
3	SZB*r*	Test bit ***p*** of ***r*** and skip {S := [S] + 1} if 0

LITERAL OR MEMORY–HELD REGISTER ADDEND PARAMETER: ADD/SUBTRACT REGISTER

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	1	0	*f*	*i*	*i*	*i*	*m*	*j*	*j*	*j*	*j*

For these orders, the parameter ***p*** is either the literal ***j*** or [M***j***].
r is: ***i***=0 → A; 1 → B; 2 → X; 3 → Y; 4 → S; 5 → P.

USERCODE FORMAT

mnem L***j*** {for ***m***=0}
mnem M***j*** {for ***m***=1}

INSTRUCTIONS

f	*mnem*	OPERATION
0	ADM*r*	Add ***p*** to ***r***
1	SBM*r*	Subtract ***p*** from ***r***

LITERAL OR MEMORY–HELD REGISTER SHIFT–LENGTH PARAMETER: 16–BIT SHIFTS

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	1	1	0	*f*	*f*	*i*	*m*	*j*	*j*	*j*	*j*

For these orders, the parameter ***p*** is either the literal ***j*** or [M***j***].
r is: ***i***=0 → A; 1 → B.

USERCODE FORMAT

mnem L***j*** {for ***m***=0}
mnem M***j*** {for ***m***=1}

INSTRUCTIONS

f	*mnem*	OPERATION
0	LSR*r*	Logical shift right ***r***, ***p*** places
1	LSL*r*	Logical shift left ***r***, ***p*** places
2	ASR*r*	Arithmetic shift right ***r***, ***p*** places
3	CSL*r*	Circular shift left ***r***, ***p*** places

LITERAL OR MEMORY–HELD REGISTER SHIFT–LENGTH PARAMETER: 32–BIT SHIFTS

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	1	1	1	0	*f*	*f*	*m*	*j*	*j*	*j*	*j*

For these orders, the parameter ***p*** is either the literal ***j*** or [M***j***].

USERCODE FORMAT

mnem L***j*** {for ***m***=0}
mnem M***j*** {for ***m***=1}

INSTRUCTIONS

f	*mnem*	OPERATION
0	LSRE	Logical shift right E, ***p*** places
1	LSLE	Logical shift left E, ***p*** places
2	ASRE	Arithmetic shift right E, ***p*** places
3	CSLE	Circular shift left E, ***p*** place

LITERAL OR MEMORY–HELD REGISTER PARAMETER: LOAD REGISTER

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	1	1	1	1	*f*	*f*	*m*	*j*	*j*	*j*	*j*

For these orders, the parameter ***p*** is either the literal ***j*** or [M***j***].
r is: ***f***=0 → A; 1 → B; 2 → X; 3 → Y.

USERCODE FORMAT

mnem L***j*** {for ***m***=0}
mnem M***j*** {for ***m***=1}

INSTRUCTIONS

f	***mnem***	**OPERATION**
0	LDMA	Load ***p*** into A
1	LDMB	Load ***p*** into B
2	LDMX	Load ***p*** into X
3	LDMY	Load ***p*** into Y

REGISTER–TO–REGISTER OPERATIONS

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	1	*f*	*f*	*f*	*i*	*i*	*j*	*j*	*j*

r is: ***i***=0 → A; 1 → B; 2 → X; 3 → Y.
s is: ***j***=0 → A; 1 → B; 2 → X; 3 → Y; 4 → S; 5 → P.
t is: ***j***=0 → ZZ, *r* = 0?; 1 → PZ, *r* ≥ 0?; 2 → NN, *r* < 0?; 3 → PP, *r* > 0?; 4 → NZ, *r* ≤ 0?; 5 → DD, *r* = –1?;
6 → IZ, *r* := *r*+1; *r* = 0?; 7 → DZ, *r* := *r*–1; *r* = 0?.

USERCODE FORMAT

mnem ***s*** {for ***f*** in 0..5}
mnem ***t*** {for ***f*** = 6, 7}

INSTRUCTIONS

f	***mnem***	**OPERATION**
0	ADR***r***	Add ***s*** to ***r***
1	SBR***r***	Subtract ***s*** from ***r***
2	LDR***r***	Load ***s*** into ***r***
3	EXR***r***	Exchange ***s*** and ***r***
4	SER***r***	skip {S := [S] + 1} if ***s*** = ***r***
5	SUR***r***	skip {S := [S] + 1} if ***s*** ≠ ***r***
6	STC***r***	Apply test ***t*** to ***r***, skip {S := [S] + 1} if **true**
7	SFC***r***	Apply test ***t*** to ***r***, skip {S := [S] + 1} if **false**

LITERAL OR MEMORY–HELD REGISTER PARAMETER: PRIVILEGED OPERATIONS

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	*f*	*f*	*f*	*m*	*j*	*j*	*j*	*j*

For these orders, the parameter ***p*** is either the literal ***j*** or [M***j***].

USERCODE FORMAT

mnem L***j*** {for ***m***=0}
mnem M***j*** {for ***m***=1}

INSTRUCTIONS

f	***mnem***	**OPERATION**
1	INTN	Interrupt processor ***p***
2	RDOM	Read domain descriptor into E from domain register ***p***
3	SDOM	Set domain descriptor from E into domain register ***p***
4	TRGA	Trigger (command) A to I/O Processor (IOP) ***p***
5	TRGB	Trigger (command) B to IOP ***p***
6	TRGC	Trigger (command) C to IOP ***p***
7	TRGD	Trigger (command) D to IOP ***p***

PARAMETERLESS: *f* IN 0..F → PRIVILEGED, *f* IN 10..1F → UNPRIVILEGED

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	*f*	*f*	*f*	*f*	*f*

USERCODE FORMAT

mnem

INSTRUCTIONS

f	***mnem***	**OPERATION**
00	HALT	Stop the ALP until restarted by the operator
01	WAIT	Stop the ALP until an interrupt is requested
02	INHI	Inhibit interrupts
03	ALLI	Allow interrupts
04	REDL	Read highest–priority MIU interrupt level into A
05	SETL	Set MIU interrupt level given in A
06	RESL	Reset MIU interrupt level given in A
07	MCTA	Move Conditions register to A
08	ATPC	Move A to Conditions register
09	ENTL	Enter level (i.e., dispatch process)
0A	SALM	Set audio–visual alarm
0B	RALM	Reset audio–visual alarm
0C	SDSU	Set status of domain descriptor in E to *unavailable*
0D	SDSR	Set status of domain descriptor in E to *read–only*
0E	SDSC	Set status of domain descriptor in E to *code–only*
0F	SDSW	Set status of domain descriptor in E to *readable–and–writable*
10	CLRA	Set A to 0
11	CLRB	Set B to 0
12	MRKA	Set A to –1
13	MRKB	Set B to –1
14	SNCO	skip {S := [S] + 1} if No Carry Overflow
15	SNAO	skip {S := [S] + 1} if No Arithmetic Overflow
16	EXEC	Executive (system) call
17	TEST	Test point trap, used to invoke a debugger
18	FLTI	Float integer in E
19	FLTF	Float fraction in E
1A	FIXI	Fix integer in E
1B	FIXF	Fix fraction in E
1C	STND	Standardise (normalise) E
1D	NEGF	Negate floating point
1E	NEGA	Negate A
1F	NEGE	Negate E

SOME NOTEWORTHY INSTRUCTIONS

JUMP

This is the only general jump order; all conditional orders merely skip the following instruction word. For example, to go to L if [A] = 0, we might use code such as:

```
STCA NZ      / skip if [A] ≠ 0
JUMP S(L–*) / self–relative jump to L if [A] = 0
```

In combination with the restricted addressing range of the Multum, the skip–and–jump idiom can lead to rather inefficient control structures. The problem lies in the jump: at the time it is generated, if the order labelled L is more than 128 words away, or a compiler does not know how far away L is, a direct jump cannot be used. This forces us to use an indirect jump, with a second word reserved for the target pointer. It also forces us to include a hop around the indirect jump, and to reverse the sense of the skip, thus:

```
STCA ZZ      / skip to the jump if [A] = 0
JUMP S2      / hop around the jump if [A]≠ 0
JUMP S0I     / self–relative indirect jump to L if [A] = 0
(L)          / pointer to L
```

Life would have been so much easier with 2–word conditional jump instructions instead of the skips:

```
JTCA ZZ      / jump if [A] = 0 (NOT Multum!) …
(L)          / to L
```

LINK

LINK is used in concert with JUMP to synthesize subroutine calls. Relatively primitive minicomputers of the era had a call order that stored the return address in the first word of the subroutine, and jumped to the second. This grievously hampered the use of re–entrant code. The Multum LINK order can place the return address in any addressable location. In particular, it can be stored in the stack frame of the called subroutine, so that recursive and re–entrant code is well supported. In general we do not know how far away SUBR is, so we must again resort to indirection:

```
LINK ZX+     / stack the return address
JUMP S0I     / self–relative indirect jump to SUBR
(SUBR)
```

But now the return address saved by LINK leads back to the target pointer word! In consequence, it is necessary to increment the return address inside SUBR before using it:

```
INCS P(RA)   / increment the stored return address
JUMP P(RA)I / go to the stored return address
```

These problems with JUMP and LINK are much less acute for the human Usercode programmer, who should have a good idea of whether a short–form jump would be sufficient, and who can make amends if the assembler finds differently. It is also fair to say that a 2–pass compiler should be able to do a much better job with jumps. Even a 1–pass compiler might be able to make worthwhile mitigations, such as marshalling target pointers into 'pools', interspersed between subroutines, so that they do not take up space in–line with the JUMP. A thorough–going implementation along these lines might yield efficiencies as good as the absent 2–word conditional jump orders, and perhaps even better if many jumps can share the same pooled pointer.

INCS,DECS

These instructions update their operand atomically, so they are useful for implementing general semaphores. They are also nice for implementing counting loops.

WAIT,INHI,ALLI

The Multum defines three classes of interrupt. **Internal interrupts** include: power failure, store parity error, invalid instruction, Executive call, and virtual store access violation. Power and parity interrupts form **class 0**; invalid instructions, Executive calls, and access violation interrupts form **class 1. Class 2** consists of the **external interrupts**: timer expiry, operator control panel key–press, and I/O interrupts. Interrupts are assigned both a priority (power failure being the highest and I/O the lowest), and a store location ('interrupt vector') in the range 20_{16} .. 27_{16} that is set to contain the address of the corresponding handler.

When a class *c* interrupt is effected, all interrupts of class *c* or greater are inhibited.

The WAIT order allows an ALP to idle, executing no instructions, and hence consuming no store cycles in useless competition with other processors. On receipt of any interrupt request, instruction sequencing resumes as normal.

The INHI instruction is used by an Executive to delay all further interrupts, typically while it is running an interrupt handler. In this state, interrupt requests are noted but not effected. ALLI undoes the effect of INHI, and would typically be used just before return from an Executive to a process level. Interrupts are inhibited for one further instruction execution after obeying ALLI, to allow time for an ENTL instruction to complete the transition from Executive to process. All of these instructions are privileged.

REDL, RESL, SETL

The Multi–level Interrupt Unit (MIU) is an option that arbitrates among interrupt requests from up to 16 external devices (typically, I/O processors) and presents the ALP with the asserted request of highest priority. In the absence of an MIU, interrupts from I/O devices are 'commoned' and share a single interrupt line. In this case, on receiving an I/O interrupt, an Executive must interrogate each active device to discover the source of the request.

In response to an MIU interrupt, an Executive can discover the identity of the requesting device by means of the `REDL` order, which encodes the position of the highest–priority asserted MIU level and delivers it to the `A` register. That value can then be used to index a further vector of device–handler addresses. `RESL` is used to reset an asserted level in the MIU. It would typically be the first action of such a handler. `SETL` is used to assert a level in the MIU, thus scheduling a 'software' interrupt for future action. Again, these are all privileged instructions.

ENTL

The `ENTL` instruction performs the switch to another process. Domain register 1 is pre-loaded with the descriptor for the master segment of the target process. `ENTL` then (re–)establishes its context by loading the values of the working registers and the rest of the domain registers from the first few locations of the new master segment. The non–virtual memory ALP/1 had no domain registers and fewer working registers, so only the latter were loaded by its more limited `ENTL` instruction.

MCTA, ATPC

These instructions copy the `C` (Program **C**onditions) register to/from `A`. `C` is a 16–bit register that defines the operating state of the ALP. Its bits are laid out as follows:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
m	*m*	*m*	*m*	*a*	0	*f*	0	*o*	*p*	*v*	*v*	*u*	*i*	*s*	*s*

where:

m is the number of a store **m**odule reporting a parity failure

a indicates a parity failure in the **A**LP during a store cycle

f indicates a **f**loating–point error, and that the ***s*** field should be interpreted accordingly

o indicates storage **o**verflow (an attempt to access a non–existent physical or virtual address)

p indicates an attempt to execute a **p**rivileged instruction in user mode

v indicates an access **v**iolation, an attempt to access a domain in an impermissible manner

u indicates whether the ALP is in **u**ser mode or in privileged mode

i indicates whether **i**nterrupts are **i**nhibited

s indicates the arithmetic **s**tatus:

if ***f*** is zero, bit E indicates fixed–point arithmetic overflow, and bit F indicates arithmetic carry

if ***f*** is one, bits E and F jointly encode one of four possibilities:

- 00 could not convert from floating point to fixed point (`FIXI` or `FIXF`)
- 01 floating point overflow
- 10 floating point underflow
- 11 division by zero

Since many of these bits are security–sensitive, both `MCTA` and `ATPC` are privileged instructions.

Note that it is possible for a process to run in privileged mode and/or with interrupts inhibited, if the corresponding bits in its `C` register are asserted. This feature was exploited to good effect in the microkernel Executive.

MULTUM INSTALLATION ROSTER

It is difficult to be sure how many Multum computers were built. There was an ALP/3 at GUCS, and there must have been at least an ALP/2 at ICS's Crewe base, because they completed an Executive with virtual memory support, which would have required extensive testing on a /2 or /3. There was also an ALP/1 there. I seem to remember that it had a drum store for efficient access to program development software. Another ALP/1 was owned by Monotype, in Redhill, presumably with a view to applications in typesetting.

Regrettably, I never photographed the ALP/3; but one of its designers, Dave Yardy, did. Visiting GUCS for a short conference of existing and potential customers, he seized the opportunity to view the configuration and take a picture. The occasion went with a bang—just as he entered the computer room, one of the large capacitors in the power supply exploded and shrapnel rattled around inside the cabinet. That fault was fixed very quickly!

MULTUM SOFTWARE

Sadly, no source code for Multum software has survived, and almost no documentation. What little remains is listed in the following Bibliography.

TOPSY

As delivered to Glasgow University, the ALP/3 was provided with a basic suite of paper–tape based software, consisting of: a text editor; a 2–pass Usercode macro–assembler that generated relocatable object code; a 2–pass relocator and linkage–editor, called Integrator; and an elementary program loader. Using these programs tried the patience: source code on paper tape had to be read into the assembler, respooled, and read again; assembly output tapes were fed twice through the Integrator in the same manner; and the integrated object program had to be respooled before being read into core by the loader. This was slow (always) and disaster prone (for the clumsy operator, like myself).

Our first priority was to exploit the disk drive to do away with as much paper tape as possible. So Iain Smith and I produced TOPSY—you can probably guess the reason for the name—which was a minuscule subset of the intended microkernel Executive. TOPSY allowed source code, object modules and loadable programs to be held in designated cylinders of the disk. A certain amount of manual intervention was still required, at the points where paper tapes were formerly respooled, but by redirecting I/O streams to the disc TOPSY hugely increased the convenience and celerity of program development. That made it practical for Pascal compiler development to be self-hosted on the Multum.

TOPSY's sole merit was that it worked. A user's guide survives, but I would be embarrassed to publish it!

PASCAL COMPILER AND PUMA

The Pascal compiler was half–bootstrapped from the #XPAC compiler for the 1900 Series. Robert Cupples used an ICL 1904A running GEORGE 3 at Strathclyde University to retarget the code generators, and then to make the retargeted compiler compile itself. The result was a very long roll of paper tape, output by the 1900 and carried to the Multum. It represented the compiler logic as a series of calls on Usercode macros. The latter implemented pseudo–instructions that were both more compact, and more convenient for the compiler, than plain Usercode. Nevertheless, it was a lot of text, and it took a long time to assemble on the Multum, even with TOPSY. The bottleneck was the macro expansion process, which—we were dismayed to discover—was glacially slow, to the point that assembling the compiler took most of a working day. This was (just about) tolerable for once–a–week compiler updates from the 1904A, but unacceptable for Pascal programs that needed to be amended, compiled and tested several times a day on the Multum itself.

A partial solution was provided by PUMA, the Pascal Usercode Macro Assembler, coded up by Ian Christie. This was a load–and–go assembler written in Multum Pascal, that generated machine code directly, and in a single pass, as fast as the compiler's output could be read. PUMA reduced the time taken to assemble and load a simple Pascal program, from many tedious minutes to a few fleeting seconds. We intended to similarly replace the Integrator with a disk-based linker, to be called LYNX, but that never got beyond the planning stage. Perhaps we should have trademarked the full range of feline product names!

THE MICROKERNEL EXECUTIVE

The main design principle of the microkernel Executive was that it should implement as much functionality as possible in virtual machines—that is, in processes—rather than in a monolithic supervisor. First–level interrupt handling, time slicing of the BIOP and the ALP, domain management, and interprocess communication (by message passing and segment sharing), were the sole work of the microkernel.

Some interrupts—for example device interrupts signalling the end of a BIOP time slice—were fully handled within the microkernel. Others—for example those signalling the completion of an entire transfer request—were handed off to interested processes in the form of messages from the microkernel. A very low–overhead message passing mechanism was designed to make this practicable. Device management, program loading, segment management, swapping, scheduling, filing systems, and job management, were all intended to be the work of privileged and/or trusted processes written in Pascal.

In reality only the microkernel, a hand-crafted set of device driver processes, and a test workload, reached completion. The prototype filing system was later reused with floppy disk drives, in an embedded system logging data from a scientific experiment; so that was not a total loss.

THE DILL-RUSSELL CONNECTION

ICS contracted a small software company, Dill–Russell Holdings, to supply it with FORTRAN and BASIC compilers. Dill–Russell engaged David Hendry, who had designed and implemented the Atlas FORTRAN V compiler. He planned to write the compilers in his low–level, machine–independent language SNIBBOL. Other members of the team included Dik Leatherdale, Tom Moran, Harry Martienson and Mary Lee, all formerly of London University's Atlas Computing Services. Dik Leatherdale was given the job of writing the runtime support routines for FORTRAN I/O, using the Monotype ALP/1 at Redhill, but was only six weeks into the job when ICS collapsed.

APPLICATIONS PROGRAMS

To the best of my knowledge, only one genuine application program ever ran on the Multum, and that was rather abstruse. It was written by my GUCS colleague, the mathematician Jenifer Haselgrove, to discover a tiling of a 15×15 square by 45 Y–pentominoes. She wrote exquisitely crafted Usercode that exploited the Multum's complex addressing modes to represent the geometric data structures; and she provided ASCII–graphical output of the filled rectangles on the console VDU or line printer. Many years later her resulting publication [4] was referenced by Donald Knuth, in a paper [5] that contains the only mention of the Multum I have ever found in the broader literature.

ACKNOWLEDGEMENTS

With thanks to John Cavouras, David Hendry and Dik Leatherdale for Multum information I had forgotten or lost.

REFERENCES

[1] J. Welsh and C. Quinn; *A PASCAL Compiler for ICL 1900 Series Computers*; Software–Practice and Experience, Vol. 2, No. 1; 1972.

[2] N. Wirth; *The Design of a PASCAL Compiler*; Software–Practice and Experience, Vol. 1, No. 4; 1971.

[3] W. Findlay; *The Performance of Pascal Programs on the MULTUM*; Glasgow University Computing Science Department Report No. 6; July 1974.

[4] Jenifer Haselgrove; *Packing a square with Y–pentominoes*; Journal of Recreational Mathematics, 7; 1974.

[5] Donald E. Knuth; *Dancing Links*; Millenial Perspectives in Computer Science, pp. 187-214; 2000; available online at `arXiv:cs/0011047`.

FURTHER READING

Many of the following can be viewed at: `www.findlayw.plus.com/Multum/Documents`

PUBLICATIONS OF GLASGOW UNIVERSITY COMPUTING SCIENCE DEPARTMENT

Multum Standard Function Memos:

• #1: A Standard Interface Convention for the 'Mathematical' Functions; W. Findlay; 11/5/1973.

Multum Pascal Compiler Memos:

• #1: A Run Time Environment for Multum Pascal; W. Findlay; 20/12/1972.
• #2: Amendments to the Run Time Environment; W. Findlay; 21/1/1973.
• #5: Code Generation; J. Cavouras; July 1973.

Multum Operating System Memos:

• #1: An Operating System for the Multum: Outline Specification; W. Findlay; 1972.
• #2: Software Implementation Languages for the Multum Computer; J.W. Patterson; 1972.
• #3: Outline Specification of a Virtual I/O System; W. Findlay; 17/8/1972.
• #4: Draft Specification of the Backing Store System; W. Findlay; 1/11/1972.
• #5: Introduction to the Virtual I/O System; W. Findlay; January 1973. {lost?}
• #6: Revised Outline Specification; W. Findlay; March 1973.
• #7: VIOS Implementation Description; W. Findlay; 16/7/1973.

Multum Hardware Memos:

• #1: Proposals for an Interval Timer in the ICS Multum Computer; W. Findlay; 20/6/1972.
• #2: Fixed and Floating Point Arithmetic; J.E. Jeacocke and W. Findlay; 17/8/1972.
• #3: Arithmetic on the ICS Multum; J.E. Jeacocke and W. Findlay; 15/9/1972.

PUBLICATIONS OF INFORMATION COMPUTER SYSTEMS LTD.

• User Specification: Executive–ALP 1.
• Usercode Language; C.A. Ashurst; 17/11/1971.
• Programmable Timer; D.T. Yardy; 9/10/1972.
• Specification of the ALP Type 3; D. Illing; 8/3/1973.