From the other side of the Alantic, How to improve upon the MU5 design.

James C. Brakefield

Abstract:

The MU5 computer and its commercial counter-part the ICL2900 have become historically interesting in much the same way that Algol 60 prompted research in programming languages. What follows is my extrapolation of where the MU5 architecture leads.

Preliminaries:

The idea undergoing evolution is that of what a memory address should be. Ie., the earliest developements where those of indirect addressing and index registers. Since then there has been continuous experimentation with greater and greater variety of addressing modes. On those computers whose word size is greater than the address size there has been experimentation with additional fields associated with the address qualifying the object referenced by the address. These kinds of address words are called descriptors.

A somewhat parallel evolution is to add non-data bits to the word to qualify the word by itself. These are called tags. There is a practical difficulty with tags in that they are inherently less memory efficient than descriptors and they do not allow qualification of less than a whole word. Finally, some computers provide both tags and descriptors.

The reason tags are inefficient is that every word must have a tag. If the word contains fields, the tag can not fully describe them. This limits the use of fully packed records. If several words all have the same tag, such as in an array, then their tags are all the same. This again is inefficient.

What makes this all the more interesting is the use of virtual memory. With large virtual address spaces complete files can be mapped into the address space of an application program. The record structure of the file can then be exploited in the program. Computer hardware which facilitates record manipulation in this environment is very handy.

This paper takes what I call an American viewpoint and restricts descriptors to describing only single objects. By this I mean that vectors and arrays are not supported directly with corresponding descriptors. The problem is that current computer architecture uses addresses of about the same size as words. Thus, to include a length or seperation field in the descriptor requires a double-precision (ie. two word) descriptor.

The most efficient approach then seems to be that single as opposed to vector operand instructions use single precision descriptors and if the computer has vector instructions, this is where to use double precision descriptors.

The architecture:

The operands for instructions shall be unsigned integers, signed integers, floating point numbers, and collections of these. The memory allotment for such operands shall be variable. Unsigned integers shall be allowed to have any power-of-two size from one to 64 bits. Signed integers shall be allowed power-of-two sizes from two to 64 bits. Floating point numbers shall have sizes from four to 64 bits. A given implementation may provide additional sizes at the expense of a smaller address space.

The choice of formats for integers and floating point is not entirely open. Conversion between signed and unsigned is simpliest if two's complement is used. Extraction of signed integers from memory and the extension of short integers to longer integers favors sign and magnitude with sign at the least significant end. The current popularity of two's complement will probably ensure its survival.

With floating point the extraction of the smaller formats from memory and extension of short immediate formats argue for having the number of exponent bits proportional to size. One easy way to do this is to interleave them with the mantissa bits on say a one-to-three ratio. This requires a reformating circuit between the registers and the floating point arithmetic unit, which may be the same unit doing integer arithmetic.

Collections of data are records. Record sizes have no upper limit. Thus as a minimum records of 4, 8, 16, 32, 64, ... bits must be supported.

Instructions shall be coded in 8 bit multiples. Descriptors referencing instructions will not be required to specify size.

The operand type and size is always coded with the address. This becomes the descriptor. A complete list of data structures is:

The unsigned integer of 1, 2, 4, 8, 16, 32, & 64 bits
The signed integer of 2, 4, 8, 16, 32, & 64 bits
The floating point number of 4, 8, 16, 32, & 64 bits
The record of 4, 8, 16, 32, 64, ... bits
The descriptor of 32 and possibly 64 bits
Program code in multiples of 8 bits

The first option is to support 3x sizes, ie. 3, 6, 12, ...
The second option is to support 5x sizes, ie. 5, 10, 20, ...
The "n"th option is to support all sizes from 1 to 64 bits

On implementations which use a 3x word size (such as 24 and 48 bit word size), both the 1x and 3x sizes must be supported. In this case, the required descriptor size will be a 3x size. Other descriptor sizes may be optionally supported.

The 1x descriptor:

There are two fields with leading one-bit coding. The type field (least significant part of the descriptor) is coded as follows:

```
        xxxxxxx1        unsigned integer
        xxxxxx10        signed integer
        xxxxx100        floating
        xxxxx000        record

        And the address/size field:
        yyyyyyy11       unsigned integer of one bit
        yyyyyy1xx       unsigned integer or signed integer of two
bits
        yyyyy1xxx       unsigned, signed, float, or record of
four bits
          .....
Examples:
        yyyyy1001       (unsigned of four bits)
        yyyy10010       (signed of eight bits)
        yy1000000       (record of 32-bits)
```

The 3x descriptor:

The least significant bit is a 3x specifier. If set, size is multiplied by three. There is one less address bit.

```
        xxxxx10         unsigned integer of 1, 2, ...  bits
        xxxxx11         unsigned integer of 3, 6, ...  bits
        signed, floating, & records similarly
```

The 3x-5x descriptor:

The 3x descriptor and 1x descriptor only allow an item of 2**n size to be on a 2**n boundary. This is a little restrictive in the larger sizes. The 3x version takes one bit, the 3x-5x version takes two bits, but there is enough codes so that an item of 2**n size can be on a 2**n-1 boundary.

```
        xxxxxxxx0       of 2, 4, ...  bits on 1, 2, ...  boundary
                xxxxxx010       unsigned
                xxxxxx100       signed
                xxxxx1000       float
                xxxxx0000       record

                xxxxx0110       bit on bit boundary
                xxxxx1110       unused
        xxxxxxx01       of 3, 6, ...  bits
                xxxxxx101       unsigned
                xxxxx1001       signed
                xxxx10001       float
```

```
        xxxx0000l       record
xxxxxxxll       of 5, 10, ...  bits
        xxxxxxlll       unsigned
        xxxxxl0ll       signed
        xxxxl00ll       float
        xxxx0000l       record
```

The instruction set:

The use of instructions with fields for operand specification is abandoned. The instuction string consists of intermixed operand specification and instruction commands. Operand specification may be arbitrarily complex, however, such complexities do not interfere with virtual addressing requirements.

The result is that instruction commands are function calls with little variability. The complexity is in operand specification which is completely removed from command specification.

Another consideration is to the register and stack facilities of a given implementation. The author's preference is to provide a limited number of general purpose registers (eight will do) and to emphasize the use of a stacking mechanism. There are to be two stacks. One holds intermediate calculations, descriptors, and subroutine parameters. The other holds return addresses, loop counters and limits, and any other control information (status bits, etc).

With this much specified, what remains is determination of the addressing modes provided. There are several general categories:

A) Operand comes from the instruction stream (immediate data)
B) Operand comes from one of the registers or one of the stacks
C) Operand is reached indirectly through a descriptor in one of the registers or from within one of the stacks
D) The descriptor in C is modified by indexing, offsetting, or indirection
E) The offset or index is arrived at by the general operand specification process
F) The descriptor is pre-decremented or post-incremented by the size of the operand

Due to the variable length nature of operand specification it preceeds the instruction which utilizes the operand. The situation is essentially that of "reverse polish notation". Any operands needed by an instruction, but not specified are taken from the value stack.

A mode:
   Immediate operand of 0..63
   Immediate operand of 8n bits where n = 1..8
B mode:

Register operand, register Ø..7
Top of value stack
Top of control stack (typically for loop count)
Operand is at an offset from one of the address registers:
either stack pointer, base or global pointer, frame pointer, offset
value specification follows in operand format

C mode:
Register holds descriptor of operand
Top of value stack holds descriptor of operand
A or B mode operand is descriptor of operand instead of operand itself

D mode:
C mode descriptor is indexed by signed integer (index is multiplied by the size of the operand and added to the address)
C mode descriptor is offset by a second descriptor, the size of the operand is taken form the second descriptor, the offset must be less than the size of the original descriptor operand
The C mode descriptor is taken as an indirect address of yet another descriptor

E mode:
The D mode index and offsets are specified by following operand specifications

F mode:
Mode flag preceeds operand specification and identifies that pre-decrement/post-increment will be applied

All of the above modes can be combined to use less than 256 codes. The remaining codes are used for instructions (op codes). An instruction such as ADD comes in three forms: single operand (increment), two operand (replace add), and three operand (add and store). Unspecified operands come from the value stack. ADD would also have an integer/floating point mode in each of the three forms. Register and stack references assume the operand is in the correct form (signed, unsigned, floating, or descriptor). References through descriptors use the form specified by the descriptor which may require automatic type conversions to match the form specified by the op-code.

Moves & format changes
Move, Move unsigned, Move signed, Move floating, Move descriptor Exchange, Clear

Unsigned, Signed, & Floating
Test, Compare, Add, Subtract, Multiply, Divide, Abs, Minus, Max, Min

Conditional branches, Looping
Branch, Jump, Jump & save, Count up/down loop, Count to limit loop, Case

Logical
And, Or, Exclusive or, Not, Masked test

Stack operations
Pop, Duplicate, Rotate up/down

References:

1. Morris, D. and Ibbett, R.N., 1979, The MU5 Computer System, Springer-Verlag, N.Y.

2. Bishop, J.M. and Barron, D.W., Aug. 1981, Principles of Descriptors, The Computer Journal, 24:3:210-221.

3. Izatt, W.T. and Schmitz, E.A., Nov. 1981, Data structures and descriptors in the ICL2900 series and beyond, The Computer Journal, 24:4:301-307.

4. Ibbett, R.N. and Capon, P.C., Jan. 1978, The Developement of the MU5 Computer System, CACM 24:1:13-24.