**TROC36\_24, 42\_24 & 48\_24[[1]](#footnote-1)**

**Using the same ISA for different word sizes**

**A Register Oriented CPU for 24, 32, 36, 42, 48 or 64-bit word size**

**James Brakefield**

**Preface**

A concise description of the digital processor families presented herein:

The register size sets the address range and the standard word size. A general purpose register file of 32 registers is provided. The processor is, for now, Harvard architecture with 16, 24 and 32-bit instructions within an 8-bit byte addressable instruction memory. The data memory is based on the register size with the byte size being the smallest sub-multiple of the word size (36: 9-bit bytes, 42: 21-bit bytes[[2]](#footnote-2), 48: 12-bit bytes). The external memory is 8-bit byte addressable and memory transfers between external memory and internal/cache memory is in 256-bit aligned chunks. Seven 36-bit words fit into 256-bits with 1.6% waste, six 42-bit words also with 1.6% waste and five 48-bit words with 6% waste[[3]](#footnote-3)[[4]](#footnote-4). Data transactions with external memory require an address divide by 7, 6 or 5 to select the correct 256-bit external memory word.

The basic ISA supports up to four data sizes and four data types. There is a register file of 32 registers, one of which is the “residue register”[[5]](#footnote-5). It serves to hold the carry and overflow from add and subtract instructions. It also gets the upper half of a multiply and the remainder of a divide. For floating point it gets the round-off error or the remainder of divide. At this time the Program Counter (PC) is part of the register file. PC and residue are distinct registers multiplexed on to the register file outputs.

The basic ISA allows for variable length immediate values encoded as needed within or following each instruction. These immediate values may be small constants, memory offsets, floating point values and memory addresses. There are single byte op-codes for maintaining half and word alignment to simplify high performance implementations.

The ISA supports three operand instructions; such as a three operand integer add or subtract. This allows use of the residue register instead of add with carry instructions. The ISA provides for data types of unsigned integer, signed integer, floating-point and a second floating-point type.

**2025 Preface**

As a result of developing a paper on floating-point, with an emphasis on latest research papers, it embraced three new definitions, a harmonized vocabulary and “new” floating point types.

**New definitions: half-bit, residue and full mantissa**

Half-bit: is an implied zero for IEEE-754 rounding, and implied one for HUB rounding and explicit for von Neumann floating-point rounding. The half-bit is the bit to the right of the normalized mantissa LSB both before and after rounding and after normalization.

Residue: A special register within the register file which receives carry, overflow, multiply upper half and divide remainder for integer arithmetic and round-off error for floating-point add/subtract/multiply.

Mantissa: Includes the hidden leading one bit, the location of the decimal point, the half-bit and the logical OR of the remaining extended mantissa.

Harmonized vocabulary: My paper has three floating-point subjects: Tapers, Rounding and Encodings

**New research papers:**

FP-Floating point, HUB rounding, recent additions to IEEE-754: Also includes my work on IEEE-754 super normal (gradual overflow, AKA symmetric tapers) and IEEE-754 symmetric adjustable tapers.

**Register tag bits:**

POSIT and FP-floats 32-bit require four additional mantissa bits compared to IEEE-754. Two additional exponent bits allows fully normalized mantissas and Euclidean distance without errors. Two type bits to indicate unsigned, signed and two floating-point formats.

**ISA changes:**

A minimal set of 16-bit operations provides for more compact code at the cost of space reserved for 40 and 48-bit instructions. 16-bit instructions leaves 25% of the 24 and 32-bit instruction op-code space available for 40 and 48-bit instructions.

Envisioned data types: unsigned values; two’s complement or sign-magnitude signed values; IEEE-754, extended 754, POSIT, FP-float and log base two floating point values. Current interest is in signed-magnitude over two’s complement and in minimal floating-point, extended IEEE-754 and 16-bit fixed point logarithmic floats over the other formats. Rounding mode set by status register or operation code.

The ISA includes means to save and restore tagged register contents to/from word aligned frame area.

**Minimal implementation:**

The minimal implementation includes a set of thirty six 16-bit instructions, with two five-bit register fields. Memory loads and stores require 24 or 32-bit instructions. Initially all instructions are 32-bit word aligned with no support for smaller size load/store (troc16\_16min). Half word aligned version (troc16\_16hwa), a half word aligned with tag bits (troc16\_16hwatag) and half word aligned with quad port register file (troc16\_16qphwa) all exist for development purposes.

**Philosophy**

The digital processor is the epiphany of the perfect machine. If well designed and implemented, which is normal, they run error free, have perfect repeatability and are not damaged by faulty programs. Calculations can be run with sufficient accuracy to limit digital noise to that in the source data or that due to the algorithms. The wear-out mechanisms are well understood and can be managed to give decades of error free operation. In contrast to mechanisms, the programmable computer can be configured via software to subsume the great variety of mechanical devices. It is capable of modeling in real-time the behavior of most mechanisms from rocket engines to everyday manufacturing.

Progress in integrated circuit technology allows operation of digital processors at billions of operations per second; many orders of magnitude faster than limited accuracy analog operations and still more orders of magnitude faster than mechanical mechanisms.

**Prologue**

DRAM memory chips are oriented for 8-bit bytes and binary addressing. Any viable computer architecture is constrained to work with these chips individually or in the form of memory SIMM modules. ASIC and FPGA internal memories are not limited to 8-bit bytes.

The design space for digital processors is vast. For today’s technology binary addressing and register file based CPUs are strongly dominate. Also dominate are two’s complement integers and IEEE-754 floating point numbers.

The ISA design is based on general purpose computing needs, op-code usage, displacement and short constant usage as reported in various studies. ISA generality is favored over special case optimization.

**The Business Case**

The use of the “right’ bit size for data improves memory capacity over, say, 64-bit data. This improves data cache performance and reduces gate count, allowing move cores per chip, lower power per core and possibly faster operation.

There are applications with well proven software that run on legacy computers that do not use 8-bit bytes and/or power-of-two word sizes. E.g. support 36 and 48-bit word size computation.

The use of 24-bit RISC instructions provides improved code density over 32-bit RISC ISAs. And the 16-bit RISC instructions, with two five bit register fields improve code density even further. 32-bit instructions are provided for three operand instructions and more options. Better code density also improves instruction cache performance.

The design herein targets the embedded market where code density, chip size and power matters. Code density is 33% better than 32-bit RISC. Compute power and chip area have a 33% to 75% advantage. At the expense of chip size and power, subsets of the design supporting 32-bit instruction alignment are compatible with high performance architectures. The x86-64 shows that the instruction alignment and instruction length do not limit performance.

The use of Harvard architecture allows the exact same machine code on all word sizes. That is, instructions use 8-bit byte addressing with instructions a multiple of 8-bits in length.

**Introduction**

This is research computer architecture with many goals venturing into several novel territories.

1. Same ISA across all word sizes
   1. For a Harvard architecture, the exact same encoding
   2. Otherwise, ISA can be remapped onto longer byte sizes
2. 16 and 24-bit instructions for better code density
   1. 24-bit: Three full five bit register fields, an immediate flag and an 8-bit op-code
   2. 16-bit: Two full five bit register fields, 32 instructions with a complete set of ALU operations using immediate values or the second register. The register data type tag bits allows the basic set: ADD, SUB, MUL, DIV, CMP, AND, OR and XOR to apply across the four data types. The residue register provides a means to elegantly handle multi-word precision and expand the functionality of the limited instruction set.
3. Variable length immediate values for better code density
   1. Currently favored: 12-bit immediate using an additional byte, four codes for immediate length and the constants -4…7 with no additional bytes.
4. Registers have a type field set on loads and on operation results
   1. Basic operations of Add, Subtract, Multiply, Divide, Compare, And, Or, Exclusive-or
   2. Operation type determined by operand types or by load instruction
5. Four data sizes supported: “byte”, half, three-quarters and full
6. Four data types supported: unsigned, signed integer, modified float and a second float format
7. 32-bit instructions for three operand operations
8. Macro capability that removes subroutine overhead and code bloat
   1. Instruction template substitutes registers avoiding in-lining or register save-restore
9. Expediting multiple issue via block header instruction that identifies instruction boundaries
10. Zero extension used for lengthening sub-word size data
    1. Due to unique formats for signed integers and floating-point numbers
    2. May not be wise for floating-point: difficult to decode register & memory contents
11. Floating-point supports unbiased round to odd
12. Floating-point underflow values maintain normalization both in memory and in registers
    1. Besides additional bits for the type code, each register has bits for floating-point needs
13. Floating point NaN code space used instead for exact floating-point values
    1. A way to support inexact floats without taking a mantissa bit
14. Residue register values vary with operation data type
    1. Generic mechanism for extended precision or extended length computations
15. Byte size is a function of word size sub-multiples
16. External memory (DRAM) is 8-bit byte oriented.
    1. Interaction with external memory takes place via a 256-bit shift register
    2. External memory address space is distinct from internal word size memory space

**ISA overview**

The 16-bit instruction contains a 5-bit operation code and two five bit register designators. For many instructions the second register designator provides immediate values. In all a basic set of instructions such that an adequate minimal processor is available.

The 24-bit instruction contains an 8-bit operation code which also indicates the instruction size, three five bit register fields and an immediate bit. If the immediate bit is set the corresponding register field is used either as a small numeric value or as the byte size of a following immediate value.

The 32-bit instruction adds another five bit register field and three option bits. Many instructions offer both 24-bit and 32-bit versions.

The irregular instructions such as relative branches, load immediate, IO port read/write, etc. vary as to the placement of fields. As a general rule the register fields do not move but in some cases are omitted.

Provision is for a full set of operation codes: arithmetic, logic, shifts, load and store, single operand transcendental, conditional branches, extract, insert, bit tests. Looping and predication mechanisms are based on Mitch Alsup’s My 66000 ISA.

**Micro-architecture overview**

At this time only a basic implementation is planned. Single cycle operation is used where possible. A 32 by 32-bit register file with two to four read ports and one write port form the core[[6]](#footnote-6). For an FPGA implementation LUTRAM is used for the register file and main memory consists of separate data and instruction block RAMs[[7]](#footnote-7).

Registers and memory are “little endian”.

The industry standard performance enhancing mechanisms of word alignment, pipelining, multiple issue, out-of-order, virtual memory and caches are all possible. Interrupt and exception mechanisms utilize the residue and PC registers of the register file which are not normally read or written as they are held in distinct registers. On an interrupt or exception the current PC is saved in a distinct register and the current frame pointer saved (optional) likewise. Access to these values is via IO instructions

Access to external memory requires the internal data memory address be divided by 7, 6 or 5 to get the external address. The remainder is used to control the operation of a 256-bit shift register which assembles or disassembles the 256-bit external memory word. This is a multi-cycle operation. For an FPGA implementation a ripple LUT based divide circuit is used operating with a ~10 LUT delay time. For a FPGA having an on-chip ARM processor, the ARM infrastructure is used to access the DRAM within the ARM virtual memory space.

Plans are to allow conditional source code implementation of instructions so LUT utilization and timing effects can be measured in detail.

**Background**

This project has a very long time in gestation. Sometime in high school I mastered binary arithmetic, Boolean algebra and the basic components of arithmetic: full adder, carry look ahead, multiplexors, multiplication using a Wallace tree and the barrel shifter.

First semester of college (1962) I started using Fortran on an IBM 1620. Soon I was also into assembly language. The university had both the 1620 and a CDC 1604. After obtaining Programmer Reference Manuals (PRM) I was soon considering improvements. The 1620 used a flag bit on each digit. One of the non-BCD digits could serve this purpose and eliminate the flag bit. This way one could use the non-BCD digits for the sign, a floating-point indicator, a record terminator and an instruction identifier. Today I have instead settled on radix-100 and radix-128 with a flag bit per digit pair. Still to be settled is as to whether general purpose or special purpose register file is better. Never went very far with 1604 “improvements” other than additional arithmetic operations.

Over the following years I was exposed to a number of computer architectures. Currently have ideas on “upgrades” to TI MSP430, PDP11/34, x86-64, DEC VAX and stack machines:

MSP430: One can use the BCD add instruction code space as an escape to a 24-bit instruction that supports five addressing modes, four data sizes and an address space set by the register length. The other original instructions remain in the 16-bit instructions. However the five address modes limit the register file to fourteen registers.

PDP11: One can use the floating-point instruction code space as an escape to a 24-bit instruction that supports twelve registers and six addressing modes, four data sizes and an address space set by the register length. The other original instructions can remain in the 16-bit instructions. However the six address modes limit the register file addressing to thirteen registers, a significant improvement over eight registers.

VAX: Limit memory access to a single operand/result per instruction. Reduce the number of addressing modes by eliminating double indirect. It would have 16 registers and a 24-bit instruction. Specification of an index register is by sixteen one byte prefix instructions.

X86: Get rid of all the prefix instructions and use a 24 and/or 32-bit instruction format offering 16 general purpose registers and the familiar x86 addressing modes.

In the last few years I have monitored the comp.arch usenet group and learned much about high performance computer design. However my roots are in the embedded world where deterministic execution speed, instruction density and limited memory size are important factors.

As a result I have pursued instruction density and settled on two main ISAs:

1. RISC with 32-registers, 24-bit instructions and variable length immediate values.
2. Stack/accumulator machine with one stack operand specified, 16-bit instruction and five addressing modes (there is encoding space for a limited number of 8 and 24-bit instructions). An instruction size of 18-bits provides a more complete ISA.

The RISC ISA is relatively stable. There are four data sizes and four data types. One of the general purpose registers (residue register) captures carries, overflows, upper half of a multiply and the remainder of a divide. 32-bit instructions support an additional register field and it is used for indexed load/stores, three-operand add/subtract, three operand multiply-add and a few well used instructions: mux, select, bit field insert, and when the residue value needs to be included.

Driven by the lack of encoding space in the 16-bit stack ISA[[8]](#footnote-8), developed the idea of using a complete set of load instructions for each data size and each data type. This simplifies the operation codes as the type of operation can be determined from the data types of the operands: Add, Subtract, Multiply, Divide, Compare, And, OR & Exclusive-Or. There is a net gain in instruction encoding space.

It was pointed out that the auto-increment and auto-decrement add pressure on register file write port(s) and is a limiting factor for high performance. So I have developed alternate addressing modes wherein an implied index register addressing mode replaces auto-increment/auto-decrement with addition/subtraction of an implied scaled index register[[9]](#footnote-9).

Another recent development sprung from a challenge: Wide register file: Initially 16 1024-bit registers that can be accessed as 256 64-bit values, 128 128-bit values, 64 256-bit values, 32 512-bit values and 16 1024-bit values. SIMD operations are assumed. For an FPGA implementation a 64 element 256-bit register file makes the most sense.

On comp.arch there are those who would prefer 36 or 48-bit words. Thirty six divides into 256 with a remainder of 4-bits. And forty eight divides into 256 with a remainder of 16-bits. The idea is that a 36-bit word size ISA would support 9-bit “bytes” and 18-bit half-words. Likewise a 48-bit word size ISA would support 12-bit “bytes”. Was there another word size possible? Yes, a 42-bit word size.

Besides the legacy support for 36 and 48-bit word size architectures there are some performance advantages to a word size smaller than 64-bits: The arithmetic function units are smaller and memory capacity increases for a given memory. Since ASICs and FPGAs readily support 36, 42 and 48 bit word sizes and the corresponding data cache memory, the remaining problem is interfacing to the 8-bit byte oriented external DRAM memory. So, picking one of these word sizes, the interface to DRAM can be through a 256-bit shift register with a 256-bit interface to DRAM and a 36, 42 or 48 bit interface to internal, typically cache, memory. A divide by 7, 6 or 5 circuit is needed where the quotient gives the DRAM memory address and the remainder gives the word location within the shift register. DRAM is much slower than cache or register memory and the divide operation effects a small additional delay in memory access time.

And finally, why not use the same ISA for each of these word sizes? My current RISC ISA is 8-bit byte oriented and implementation would use a standard instruction cache and a standard interface to DRAM. The data side of each implementation would have 9, 21 and 12-bit bytes in a straight binary address space. Harvard architectures for 36, 42 and 48-bit word size computers that can share the exact same instruction binaries[[10]](#footnote-10)!

An FPGA implementation can now begin.

**Appendix A: Goals in more detail**

*This is research computer architecture with many goals venturing into several novel territories.*

1. Same ISA across all word sizes
   1. For a Harvard architecture, the exact same encoding
   2. Otherwise, ISA can be remapped onto longer byte sizes

The set of operations is that of necessary and useful functions. There is reserve op-code capability with the “40 and 48-bit” instruction formats. Due to the use of variable length immediate values, providing longer immediate values for larger word sizes maintains the ISA character.

1. 24-bit instructions for better code density
   1. Three full five bit register fields, an immediate enable bit and an 8-bit op-code

The use of 16-bit “compressed” instructions usually entails reduced operation codes and reduced register set. Here the most common operations have a 24-bit encoding simplifying code generation.

1. Variable length immediate values for better code density
   1. Currently favored: 12-bit immediate using an additional byte, four codes for immediate length and the constants -4…7 with no additional bytes.

Other encodings of immediate size are possible and for longer word sizes necessary. Some reduction of the short -4…7 values is necessary to support more than four byte long immediate values.

1. Registers have a type field set on loads and on operation results
   1. Basic operations of Add, Subtract, Multiply, Divide, Compare, And, Or, Exclusive-or
   2. Operation type determined by operand types or load instruction

The Boolean and some other operations ignore register data type. Also some combinations of mixed type operands may not be supported, which then cause an unimplemented instruction fault.

1. Four data sizes supported: “byte”, half, three-quarters and full
   1. 32-bit word: 8, 16, 24 and 32-bit data
   2. 36-bit word: 9, 18, 27 and 36-bit data
   3. 42-bit word: 21 and 42 bit data
   4. 48-bit word: 12, 24, 36 and 48 bit data
   5. 64-bit word: 8, 16, 32 and 64 bit data; additional data sizes via fewer short constants
2. Four data types supported: unsigned, signed integer, modified float and a second float format.

Two floating-point data types enables use of various experimental floating-point formats. The author advocates for a different format for denorms, round to half-odd and a different format for Posits. Signed integer can be the usual 2’s complement or sign and magnitude. The operation of the residue register is such the multi-word or extended precision software is straight forward.

1. 32-bit instructions for three operand operations

These instructions have an additional three instruction bits that can be used and are intended to be used for residue register update enable and sign flip on two operands. The 32-bit single operand instructions have six additional instruction bits that currently are allocated to: immediate operand, result data type, result data length, and operand sign flip.

1. Macro capability that removes subroutine overhead and code bloat
   1. Instruction template substitutes registers avoiding in-lining or register save-restore

One of the weaknesses of RISC architecture is lengthy and time consuming save/restore of registers prior to and after executing a subroutine. This is countered by in-lining short subroutines leading to code bloat. Alternatively rearranging the registers to place the subroutine operands, results and temporaries in registers 1 thru N where N can be up to 6. As an alternative for short subroutines is to “remap” the subroutine parameter and results to the registers to where they reside. Basically the same as in-lining where the subroutine references registers one thru N but the execution of the subroutine uses the one to six registers embedded a template instruction. The template instruction is an otherwise unused op-code dynamically allocated in the macro table. The macro table has instruction length, six slots to hold the new register numbers, the starting address of the subroutine and a place for the return address. Without a push-down stack for return addresses and register numbers recursion is not possible, or at least, not simple and fast.

1. Expediting multiple issue via block header instruction that identifies instruction boundaries

A high performance design, in particular for a multiple issue implementation, prefers instructions to start on word boundaries. Lacking that, having a simple way to determine where each instruction begins. Taking 256 bits as an instruction block size and requiring that there always be an instruction at the block start it is possible to have a 24-bit instruction which shows where each of up to ten following instructions start. Envision that for every three bytes there is an instruction decoder and ALU. Each instruction decoder can get its instruction from one of three places or there can be no instruction at any of the three locations. So ten two bit fields can specify the relative locations of the instructions for each of the decoders. By using sixteen op-codes the 20 bits of info fits into a 24-bit instruction. Furthermore, if there is no such instruction at the block boundary the processor falls back to less simple/slower decode.

1. Zero extension used for lengthening sub-word size data
   1. Due to unique formats for signed integers and floating-point numbers
   2. May not be wise for floating-point: difficult to decode register & memory contents

Twos complement is normally used for negative integers and requires sign extension to lengthen short integers. For zero extension of signed integers it is necessary to use sign and magnitude representation. Lengthening is by zero extension, somewhat simpler than sign extension. Sign-magnitude has two codes for zero. The second code for zero can be reinterpreted as missing, omitted or not valid that often occurs in statistical data.

For floats there is also a format which utilizes zero extension. It requires bit reversing the mantissa. And to provide additional exponent bits on larger mantissas, interleave mantissa and exponent bits. Using sign and magnitude for both exponent and mantissa completes the encoding (both signs at the LSB end). However without proper IDE support, translating between the memory binary and the floating-point value is difficult. So it is more useful to provide reformatting circuits to lengthen float data thereby avoiding mantissa bit reversal and exponent-mantissa interleave.

For both sign-magnitude and non-standard floats and Posit it is necessary to have specialized comparison circuits.

1. Floating-point supports unbiased round to odd
2. Floating-point underflow values maintain normalization both in memory and in registers
   1. Besides additional bits for the type code, each register has bits for floating-point needs
3. Floating point NaN code space used instead for exact floating-point values
   1. A way to support inexact floats without taking a mantissa bit
4. Residue register values vary with operation data type
   1. Generic mechanism for extended precision or extended length computations

For an unsigned addition or subtraction there is only a single carry bit. For two’s complement addition and subtraction there is usually an overflow indication. It is simpler to consider that for signed addition and subtraction there is an overflow or carry of: zero, plus one or minus one. So for unsigned add/subtract the residue register gets zero or one. For signed add/subtract the residue register can get 0, +1 or -1 with no need for an overflow flag; instead one checks the residue register for zero or not.

For multiple word add/subtract the three operand add/subtract can include the residue register and thus there is no need for the add-with-carry or subtract-with-carry/borrow instructions. For the situations where there are three full size operands, the overflow result can exceed +1 or -1. The residue register simply captures the resulting overflow.

For multiply the situation is also direct: the residue register gets the upper half of the product. For signed multiply both halves have the same sign.

For unsigned divide the residue register gets the remainder. The remainder can then be utilized for extended precision by the three operand divide instruction. For signed divide the situation is more complicated. However, for the sign-magnitude format with the remainder the same sign as the dividend it all works out. The three operand divide instruction concatenates the remainder in front of the dividend. The circuit must correctly handle the situation where the remainder and the dividend are of different signs.

For floating-point and Posit arithmetic the details are long and tedious. The plan is that the residue register capture the rounding error. And that this can be used to support extended precision arithmetic, primarily through the use of the multiply-add instruction and various rounding modes. The 40-bit instruction has sufficient spare bits to directly encode a specific rounding mode without use of a status/configuration register (which would be accessed via I/O instructions).

1. Byte size is a function of word size sub-multiples
2. External memory (DRAM) is 8-bit byte oriented.
   1. Interaction with external memory takes place via a 256-bit shift register
   2. External memory address space is distinct from internal word size memory space

**Appendix B: Microarchitecture**

Specialized comparators for sign-magnitude and float-Posit: Mostly a matter of routing bits to an unsigned comparator in the proper order. In addition checks need to be made for NaNs and other special cases.

Three input MAX, MIN and MEDIAN: Three comparators followed by a 3:1 MUX. Choice of residue result is TBD.

Divide: For unsigned divide, first form a 3X multiple of the divisor. Then a divide stage of three subtractions of 1X, 2X and 3X followed by a 4:1 MUX. Repeat the subtractions on the MUX output for a four bit reduction of the dividend per clock cycle (a total for six subtract circuit and two MUXs). Thus eight clocks for a 32-bit dividend. For sign-magnitude and float-Posit divide use the same process with the shorter dividend and one fewer clock for Posit and two fewer clocks for floats.

Trig-metric functions in rotations: Avoids the accuracy problems of range reduction for large angles. Take the fraction of the float-Posit and compute the result. For very large angles, the reduced angle is zero. A zero angle may not be appropriate, especially for in-exact operands, in which case a NaN could be returned?

Single cycle operation: It is expected that “simple” operations (add/subtract, Booleans, shifts, insert, extract) can be done in a single clock cycle. A multiply will likely take two clock cycles. A divide will take seven to nine clock cycles. Floating-point/Posit ADD, SUB and MAC are expected to take several clock cycles.

Load and store: With a single block RAM for both program and data memory loads and stores take an additional clock cycle. The dual port capability of block RAM is used to read/write the current and next location along with instruction and data shifters makes for a single clock cycle operation for any data or instruction alignment. Floats and Posits undergo a un-pack on load and a round and pack on store. Part of the un-pack is to adjust the exponent via a trailing zeros count on the mantissa (for denorms and Posit). Incorporation of cache memory will likely further lengthen load/store timing.

1. For TROCnn\_tp, nn is the word size, t is the number of tag bits and p is digits for instruction byte sizes. ROCnn\_0p is the name for the variant that does not have a data type code for each register in the register file. [↑](#footnote-ref-1)
2. Twenty one is not evenly divisible by two. Is it possible to support a 10-bit byte size? [↑](#footnote-ref-2)
3. It can be argued that 40 and 48-bit word sizes would accomplish the same goals without departing from an 8-bit byte size, and would give a seamless interface to external memory. Indexed loads and stores would require multiply by five or three plus scaling by a power of two. Such is faster than divide by small odd primes. And there would be no memory wastage. Due to the support of four or more data sizes in the ISA, such data sizes could be supported for 32 or 64-bit word sizes or vice versa? The issue of unaligned loads and stores remains and is present in all designs herein. [↑](#footnote-ref-3)
4. The Burroughs computers used a 48-bit memory word each with three tag bits. Fifty one divides into 256 with one unused bit. [↑](#footnote-ref-4)
5. Called “carry” register in Mitch Alsup’s My 66000. [↑](#footnote-ref-5)
6. The Residue register requires a second write port. Implementation is as a distinct register updated as required. On register reads the Residue register is multiplexed as needed into the read port results. [↑](#footnote-ref-6)
7. At this time it is planned to use the dual-port feature of block RAMs to support unaligned loads, stores and instruction fetch with single cycle performance. [↑](#footnote-ref-7)
8. Now named OSMnn\_mm (One operand Stack Machine). The first implementation is likely to be OSM18\_18 [↑](#footnote-ref-8)
9. For many of the inner loops using vector or matrix operands a single index register suffices. For ROC\_nn\_mm the longer length load/store instructions support a general purpose index register and additional scaling. [↑](#footnote-ref-9)
10. Elementary function evaluation will need optimal polynomials and polynomial coefficients on different word sizes for maximum accuracy etc. [↑](#footnote-ref-10)