ASTOC ISA Design

James Brakefield

ASTOC stands for **A**ccumulator/**St**ack **O**riented **C**omputer. It is a hybrid between Dual Stack and Accumulator architectures. Additionally it maps the stack and frame areas onto a register file. And it supports PDP-11 style memory addressing.

The ISA choices herein create a unique design providing very good code density, low subroutine overhead and an easy to follow register allocation scheme. Instructions can be one or more bytes long. Single byte instructions are the usual dual stack operators. Two and three byte instructions support referencing into the stack and frame areas for one or more operands. Additionally the two and three byte instructions provide indirect addressing into main memory.

One can concentrate on the two byte instructions and examine the various instruction fields that can exist. There is a trade-off on the capabilities of each field and what can fit into two or three bytes. Solutions for greater capability are to increase the size of instruction "bytes" or redistribute the larger fields into longer instructions.

Contents:

The instruction fields
The stack reference fields
The addressing modes
The data sizes
The replace bit
The pop/push bit
The return bit
Dedicated registers
Residue register update enable bit
ISA design
Subroutine with frame interface
Summary perspective
Glossary
Instruction fields
Eight bit byte "skinny" instruction formats
Ten bit instruction-byte "fat" formats
Op-code lists

The instruction fields:

The op-code, typically ~32 op-codes is the minimum for a complete ISA. Categories are integer operations, floating-point operators, load/store from memory and control flow.

The stack reference fields:

There are four register file reference areas: near the top of the data stack, near the top of the return stack, a frame area and a global area. Two bits suffice for the area designator and two to five bits suffice for the offsets from the top or start of the various areas. One allocation that appears adequate is access to 16 frame registers, eight data stack registers and four each return stack and global registers. The frame area can be part of either the data stack or the return stack.

The global area stays fixed and provides a place for frequently used thread or global values. The Frame area is allocated by subroutine calls and previous frame area restored on subroutine exit. The return

stack is used for return addresses, previous frame pointers and for loop variables. Forth Stack Machines frequently move items between the data and return stacks. Here the need is much less as the frame area organizes temporary variables. The data stack retains its role from Forth and other stack machines. Offset addressing into any of the four areas provides convenient and compact access to working values.

One or more combinations of data size and addressing mode (SM fields) can be used to indicate the use of the DN field as an immediate operand. DN is usually five bits and as such of the 32 codes some can be used to indicate a specific value and some can be used to indicate the number of bytes following that constitute the immediate value.

The addressing modes:

Using the registers in the register file, accessed via offsets from the four areas, as memory addresses, one can have an instruction field for memory addressing mode: register reference, register indirect reference, register indirect with post increment by data size, register indirect with pre-decrement by data size, an immediate value from within the instruction or a register indirect with immediate offset.

The data sizes:

One typically needs up to four data sizes. On a 32-bit machine these could be 8, 16, 24 or 32 bits. On a 64-bit machine these could be 8, 16, 32 and 64 bits. If data byte size is something other than eight, other possibilities occur. So use a one or two bit data size field in the two byte instruction.

The replace bit:

An accumulator machine typically offers the operation result going to either the accumulator or to the operand location.

The pop/push bit:

The accumulator can be considered the top of the data stack. It is very useful to provide a bit to indicate a pop or push of the accumulator. If the replace bit is set, the pop/push bit invokes a pop of the data stack. If replace bit is clear, and the pop/push bit is set, the previous accumulator is pushed to the data stack and the operation result goes to the accumulator.

The return bit:

Stack machines with bye or longer instructions often offer a return enable bit that pops the return stack into the PC. One can then have one instruction subroutines. From an architecture standpoint this is getting a little more work out of a single instruction. The trade-off on having a return enable bit depends on the average subroutine length. For best code density short subroutines favor the return enable bit and longer subroutines favor a one byte return instruction.

Dedicated registers:

The accumulator as a separate register whilst being considered at the top of the data stack is useful for a one clock cycle SWAP operator. A dedicated PC register can be considered as the top of the return stack. Not clear at this time if this merged role for the PC is necessary or useful. The Residue register is used to hold carry and overflow results from integer operations. It is also used to capture the upper half of a multiply or the remainder from a divide. For floating-point operations it captures the round-off error.

Residue register update enable bit:

Given sufficient instruction width, the residue-register can have an update enable bit.

ISA design:

Nine instruction fields are described above. The size of each field and where they are located within a two or three byte instruction is a design trade-off decision. Currently there is a "skinny" ISA with two data sizes and eight bit bytes. Most of the ISA is squeezed into two bytes. The "fat" version supports four data sizes, a range of offset addressing modes with various offset sizes and a greater number of single byte "stack" op-codes.

Usage data is necessary to make the best decisions on instruction field sizes, if they are even necessary and where within an instruction they are located. The Livermore Loop code fragments show the usefulness of a frame area of ~16 registers and the register indirect auto-increment addressing mode.

Subroutine with frame interface:

Prior to calling the subroutine space is reserved on the data stack for the results. This is followed on the data stack by the subroutine parameters. The CALLF instruction then preserves the previous frame pointer and where the stack pointer should point after the corresponding return. The CALLF instruction contains a byte that indicates the number of return results and the number of parameters and a bit to indicate parameters on either the data stack or the return stack. The parameter area could be placed on the return stack, however since most subroutines have more parameters than results, it is more efficient to have parameters placed on the data stack.

Summary perspective:

The data stack traces its lineage to expression evaluation. The return stack traces its lineage to support for recursive subroutines. One can argue whether the two stacks are both necessary and/or whether they can be merged? RISC architectures use the programming language compiler to allocate the registers in the register file and to manage register save/restore to/from memory. The author prefers hardware automatic stack spill and refill with straight forward register allocation compared to the non-transparent compiler allocation of registers. Additionally, the two stack mechanism provides a simple and efficient subroutine interface. The tendency of pipelined RISC architectures is for longer subroutines and in-lined subroutines where possible. Indications are that the ASTOC architecture will have twice the code density of RISC architectures. It is hoped that the savings on instruction caching

and pipeline widths for ASTOC will provide enough chip real estate to implement various speedup techniques (multiple issue and out of order completion) to match RISC performance.

The addition of the indirect addressing modes solves a problem of stack machine ISAs: How to efficiently and conveniently handle memory access in a single instruction. And how to incorporate access to subscripted arrays. The vast majority of programming language code can make good use of (DN++) and (DN+offset) addressing modes.

Glossary:

DS: Data stack, useful for the evaluation of expressions

Frame: A portion of RS or DS used for subroutine parameters, results or temporaries

Register file: A numbered set of word or address size registers accessed by register number. For FPGA implementations, multiple read address ports and a single write address port design is preferred. The ASTOC ISA is arranged such that a single register-file write port suffices for almost all instructions. If multiple writes are required additional register-file clock cycles are necessary.

RS: Return stack, useful for return addresses and looping parameters/values

Stack: A register file pointer location acts as the top of a register based stack. Offsets from the register file pointer allow access to other registers near the top of the stack. Stack pop or push is arranged by decrementing or incrementing the pointer.

Stack reference: A register file reference (a data stack, return stack, frame or global area reference) or if it is an indirect reference, the register indirect memory location. The referenced location can be read, written, read/modify/write or executed (subject to any memory protection mechanism such as read only, execute only, etc).

Stack spill/refill: Hardware or an interrupt mechanism that detects the lack of space for additional stack entries and removes and saves stack entries from the other end of the stack to make more room, transparently to the operation of the program. Refill occurs when there are too few stack entries in the register file.

TOS: Top of the data stack.

Instruction fields: The following single character bit field names provide a visual means to display the instruction formats.

X Op-code bit
F Residue-register update enable
r Subroutine return enable bit (pop return stack into PC)
P Push/pop TOS
If results go to TOS (R is 0), P is enable to replace TOS with results otherwise push TOS
If results go to stack reference (R is 1), P is pop TOS enable

R Results to stack reference or to TOS

S Data size code

M Addressing mode (stack ref, indirect ref, indirect ++ ref, indirect – ref, indirect+offset ref)
S and M fields can be combined (such as to support three data sizes and five address modes)

DN Register-file pointer selection and a short offset to that pointer

V Swap immediate with other operand

n Small immediate value. The five bit immediate code indicates a value between -8 and 15 or the byte count of a following immediate. An alternate encoding is -8 to 7, -2048 to 2047 using the next instruction byte or a byte count of a following immediate.

Eight bit byte "skinny" instruction formats (little endian):

One byte stack mode instructions:			xxxxxxxx	16 total
Two byte instructions:		smmddnnn	rPRxxxxx	N is 0 to 7
Two byte with immediate:		smmnnnnn	rPVxxxxx	Immediate value operand
Two byte control (BR, BRZ, BRNZ, LDI):		nnnnnnnn	nnnxxxxx	N is -1024 to 1023
Three byte instruction:	ffffxsss	nnnnnnnn	nnnxxxxx	CALLRF
Three byte instruction:	ffffxsss	xmmddnnn	xxxxxxxx	CALLF
Three byte instructions:	xxxddnnn	smmddnnn	rPRxxxxx	Adds second stack reference and three op-code bits

Ten bit instruction-byte "fat" instruction formats (little endian):

Fat instruction bytes are six per 64 bits. The two least significant instruction address bits are never 11.

One byte stack mode instructions:		fxxxxxxxxx	
Two byte instructions:	ssmmmddnnn	frPRxxxxxx	N is 0 to 7
Two byte with immediate:	ssmmmnnnnn	frPVxxxxxx	Small immediate value
Two byte control:	nnnnnnnnnn	nnnnxxxxxx	N is -8192 to 8191

"Skinny" Op-code List:

One byte: RTN, BRK, NOP, DUP, DROP, SWAP, ILL0, ILL1, four loop setup and four branch to start of loop
LOPW: while loop, address of loop start pushed to return stack by MLOPW one byte instruction.
LOPCTDN: count down loop, address of loop start and count pushed to RS by MCTDN instruction.
LOPCTUP: count up loop, address of loop start, limit and start count pushed to RS by MCTUP instruction.
LOPCINIT: count to loop, address of loop start, limit, count delta & start count pushed to RS by MCNTO.

Two byte control: BR, BRZ, BRNZ, LDI, CALLR, CALLRF
Two byte operations: LDB, LDSB, STB, STW, ADD, SUB, MUL, DIV, AND, OR, XOR, CMP, FADD, FSUB, FMUL, FDIV, FCMP, EXTRCT, EXTRCTS, ASR, LSR
Three byte operations: INSERT, ADD, MAC, DIV, MEDIAN, LSL, LSR, ASR, SELECT, MERGE, MOV2, MAX, MIN, CALLRF, CALLF, LDX, STX, BITBR, AND, OR, MUX. Operand negate/complement bits are available for some operands. Three operand versions of two byte instructions provide access to the Residue Register. A work in progress.