A Four Stack Processor

Bernd Paysan

25th April 2000

Abstract

This article presents an experimental architecture of a four stack/dual data move CPU. Such a four stack machine was proposed in spring 1993 in the article thread "Stack machines and RISC" in `comp.arch` to allow parallel stack processing. A complete ISA, interrupt strategies and some sample code is presented.

1 Motivation

Superscalar processors exploit the inherent parallelism of a sequential code. However, scheduling has to be done while executing. The instruction grouper has to avoid and resolve data conflicts at runtime, although they are (almost) statically known at compile time. Scoreboards, comparison areas, pipeline bypasses and history buffers (for speculative execution) consume die area and time (at best, they increase latency). Multiported register files (especially multiple write ports) increase complexity and therefore slow down register access.

The goal of this work is to reduce possible collisions in a superscalar unit by separating write pathes. Stack-like register files exploit the implicit addressing modes of stack machines to decrease instruction length. Short instructions reduce memory bandwidth and improve cache usage (longer programs fit into the same cache size). Procedure calls are cheap on stack machines; making HLL and modern operating systems faster.

Arithmetic expressions execute naturally on stacks; although stack machines suffer much more from inherent parallel, but sequential executed instructions than register machines, it is assumed that most programs use only a small number of tight coupled parallel instruction streams (up to four). It is assumed furthermore, that the results of these parallel instructions will be used by any of the following instruction, such that easy result forwarding is required.

Because branch instructions are likely to occur more frequently in such a VLIW instruction set, pipelining should be reduced as much as possible (by simplifying instruction decode); and pipelined execution should be visible to the code generator where it can't be avoided

(no stalls, no pipeline bubbles). It is proposed, that an implementation of the ISA has a three stage pipeline: fetch, decode and execute. Thus speculative decode (aborted at the end of the execution phase of the branch instruction) in both control flow branches allows "zero cycle" branches.

Thanks to Piercarlo Grandi, who had the initial idea and to Kyle Hayes for discussing many aspects of the design.

2 Implementation

The proposed machine uses four stacks cached in four LIFO latch files. Each stack has its own ALU. The four stack locations from the top of stack may be accessed by any other ALU (four read ports). The next four stack locations may only be accessed by the stack's ALU itself (one read port). The stack items below are (usually) unaddressable. This allows to choose from 32 entries each cycle (each of the four stacks can choose one of eight entries), and up to four recently computed results from any other ALU each. Each stack spills automatically on overflows and refills on underflows.

In addition to the four ALUs, two memory units allow parallel loads and stores (found be very useful in DSPs like the MC 56K). To avoid conflicts, one of the memory units reads and writes to even stacks, the other to odd stacks (stack numbers from 0 to 3). Each unit uses four register sets; each register set allows to form and modify one address. The stack paradigm isn't used in these units, because memory accesses are expected to be more global.

I chose some architectural features good for (integer) digital signal processing, because these applications become more and more mainstream ("multimedia": JPEG, MPEG and music/voice compression). So each memory register set can be used as indexed array, as ring buffer (FIFO or array with bound check), as stack (LIFO) and (for FFT) as bit reverse addressed ring buffer.

DSP algorithms require a fast multiply, shift and add unit. As this unit would be the critical path of the instruction set, it is implemented as a two cycle, visible pipelined instruction. Only two multiplication units are provided, one for the lower half of the stacks, the other for the upper half. Each stack half can issue one multiplication per cycle and finish it in the next cycle. Splitting the multiplication instruction into two halfs allows to use many modifications like multiply, add and round. I hope aggressive carry look ahead will allow to implement a fast enough multiply unit.

As ALU speed limits the overall CPU speed (superpipelining is not possible), I expect a clock rate of about 100 MHz using current 0.6μ HCMOS or 0.8μ BiCMOS processes.

3 Instruction Set Architecture

The machine is a 32 bit machine. Instruction word length is 64 bit.

Registers per stack:

s0 (TOS), s1 (NOS), ..., s7; sr (status), sp (stack pointer); if preceded by a number from 0-3: select other stack.

Registers per address unit:

R0–R3; N0–N3; L0–L3; F0–F3; each R_n has a triple of N, L, F associated (N is distance, L is limit, F are usage flags).

I'll give you the inner loop of a complex fraction FFT as example of the instruction encoding (lines beginning with ;; are comment lines). `pick` reads the source operand and pushes on the current stack. `mul` starts a multiplication $source * TOS$ and writes the intermediate result into one of two multiplier latches, depending if stack is 0 or 1 resp. 2 or 3. mul@ and variations read these latches and thus push the result on the current stack. TOS is always an implicit operand and destination, thus not encoded in the instruction.

```
;; st0:      st1:      st2:      st3:      even data        odd data
;; --        --        n         --
                       index!              do               L$loop
;; fr        hi        fi        hr

   pick 3s0  mul 0s0   pick 1s0  mul 0s0   ld 0&2: R1       0 #
;; fr hr     --        fi hi     --

   mul 2s1   mul@      mul s1    mul@
;; fr gi     2:hifr/2  fi gr     2:hrfr/2

   asr       mulr@+    asr       -mulr@+   0 #              ld 1&3: R1 +N
;; fr gi/2   (hifr+fihr)/2=:qi
;;                     fi gr/2   (hrfr-hifi)/2=:qr

   add 1s0   subr 0s0  add 3s0   subr 2s0  st 0&2: R1 N+    st 1&3: R1 N+
;; fr gi/2-qi gi/2+qi  fi gr/2-qr gr/2+qr
L$loop:
```

Two things have to be explained: First, there is a load delay of one cycle. So the load "ld" instruction takes two cycle. Data is available at the begin of the second instruction after the load. This allows to split address computation and bus/cache operation into two separate cycles. Certainly wait states must be inserted in case of a cache miss. There is no such delay for stores "st". The store takes the result of the current cycle. The numbers

like 0&2: select the destination stack(s), two numbers mean two consecutive loads. +N adds the index N_n to the address register to form the address, N+ modifies the register R_n after the address computation.

The implicit addressing of the multiplier result: unlike other operations, the multiply issue instruction leaves its intermediate results in the pipeline latches of the multiplier unit, not on the stack. So the multiply result fetch instruction needs not to operate on the same stack as the issue instruction. This allows to move the multiplier's result from one stack to the other (only inside one stack half) and eases scheduling.

The do loop statement is a Fortran-like hardware do loop without any per loop overhead.

The resulting code is quite good, compared with specialized DSPs as the MC 56K: Using two pipelined 2 cycle multipliers, each FFT step requires 4 cycles. The DSP MC 56K has one 2 cycle multiplier (not pipelined) and finishes one FFT step after 12 cycles (min).

FFT has a very high inherent parallelism — a 1024 point FFT may be split into 512 parallel elementary FFT transformations; so FFT is a typical peak rate benchmark; even for SIMD machines. However, this coding doesn't unroll the loop — it doesn't need that aggressive coding.

I'll give a much worse example with very little inherent parallelism: a walk through a list to find the list's end.

```
;;    list    0 list    --  --
last: nop     pin s1    nop nop  ld 0: s0  NEXT #
      nop     pick 0s0  nop nop  br 0 ?0<> last
;;    rubbish last 0    --  --
```

This loop speculatively loads the next list entry before checking for NIL; it would take one extra instruction to prevent this. Compiler idioms (like `while(*b.next) b=*b.next;`) could help to code this instructions (because I don't know of any compiler that inserts speculative loads). A slight modification of the loop allows to count the list with no overhead. However, the NIL load should not result in an exception.

It's even possible to search a keyed element in a list without any additional overhead:

```
;;      list    0         key <>key  --
search: nop     pin s0    sub s1     nop   ld 0&2: s0  NEXT #
        nop     pick 0s0  nop        nop   br 0&2 ?0<> search
;;      rubbish find/0    key rub.   --
                                           br 1 :0= notfound
```

4 Programmer's Model

Register file:

Stack 0	Stack 1	Stack 2	Stack 3
0s0	1s0	2s0	3s0
0s1	1s1	2s1	3s1
0s2	1s2	2s2	3s2
0s3	1s3	2s3	3s3
s4	s4	s4	s4
s5	s5	s5	s5
s6	s6	s6	s6
s7	s7	s7	s7
sr	sr	sr	sr
sp	sp	sp	sp
ip, index, loops, loope			

Bits in sr: `STAD0000|IIIIIIII` (global state) | `CCCCCCCC|0UMMX00C` (per stack state)

- S supervisor state
- T trace mode
- A address mode for instructions (0=32, 1=64 bits)
- D activates external debugger
- I interrupt disabled
- C shift count (signed)
- U shift mode: 0=unsigned, 1=signed
- M rounding mode: 0=to nearest, 1=to zero, 2=to ∞, 3=to $-\infty$
- X conditional execution (0=execute, 1=don't execute)
- O overflow
- C carry
- 0 reserved and must be set to 0

The lower half of sr is local to each stack, the upper is global for all stacks.

Memory register file:

Even stack unit			
R0	N0	L0	F0
R1	N1	L1	F1
R2	N2	L2	F2
R3	N3	L3	F3

Odd stack unit			
R0	N0	L0	F0
R1	N1	L1	F1
R2	N2	L2	F2
R3	N3	L3	F3

Flags in F_n: `00000000|00000000|00MMMMMM|000SBROZ`

Z scale displacement (N, s0, constant offset) by size
O byte order (0=big, 1=little endian)
R bit reverse addition
B create bound crossing trap
S stack usage: negate N on writes
M mask bits, if the operation crosses the limit, the lower n bits are masked out

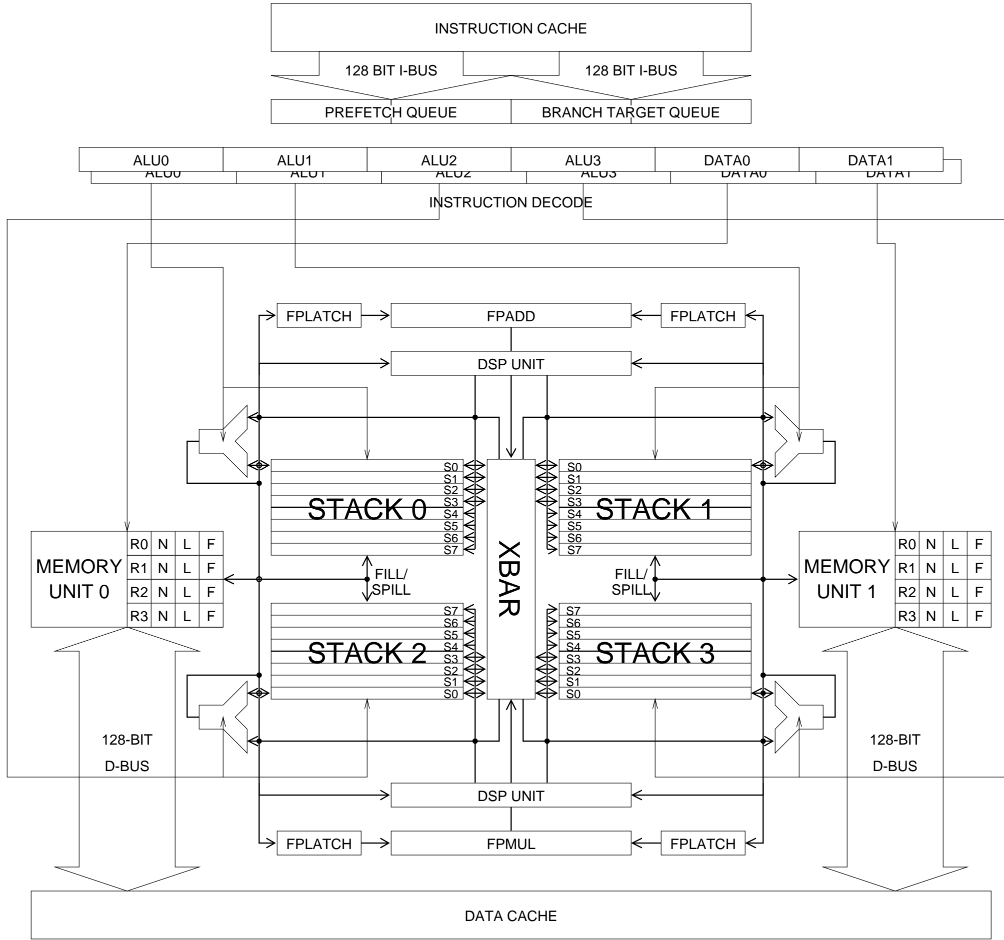

Figure 1: Data pathes

5 Instruction Format

To reduce the article's length, I introduce a short instruction notation, that is (in EBNF):

$[\langle n\rangle``*"]\ \{\ [\langle\text{static bits}\rangle]\ [\langle\text{instruction name}\rangle]\ [``\langle"\langle\text{table}\rangle``\rangle"]_{\langle\text{part size}\rangle}\ \}^*_,\ ``="\langle\text{explanation}\rangle$

All constants are in hex.

General instruction encoding (big endian):

$4 * \langle$stack code$\rangle_{10}$, $2 * \langle$data move$\rangle_{10}$, $2 * \langle$move/modify$\rangle_1$, 0_2= continue
$4 * \langle$stack code$\rangle_{10}$, $4 * (\langle$copy/pop$\rangle_1$, $\langle$flag$\rangle_4$), 0_2, 1_2= conditional setup
$4 * \langle$stack code$\rangle_{10}$, $\langle$or/and$\rangle_1$, $\langle$stack map$\rangle_4$, $\langle$copy/pop$\rangle_1$, $\langle$flag$\rangle_4$, $\langle$offset$\rangle_{11}$, $\langle$likelihood$\rangle_1$, 2_2= br/do
$3 * \langle$stack code$\rangle_{10}$, 0_1, $\langle$call/jump$\rangle_1$, $\langle$address$\rangle_{29}$, 0_1, 3_2= call/jump relative
$3 * \langle$stack code$\rangle_{10}$, 1_1, $\langle$call/jump$\rangle_1$, $\langle$address$\rangle_{29}$, 0_1, 3_2= call/jump absolute
$\langle$address$\rangle_{61}$, 1_1, 3_2= far call

stack code:

0_1, $\langle$mode$\rangle_2$, $\langle$group$\rangle_2$, $\langle$extend$\rangle_5$

extend depending on mode and group:

		group			
		0	1	2	3
mode	0	or T1	add T1	addc T1	mul T1
	1	and T1	sub T1	subc T1	umul T1
	2	xor T1	subr T1	subcr T1	pass T1
	3	T2	T3	T4	T5

T1:

0_2, $\langle$#n$\rangle_3$= small immediate constants #n:
0, −1, $7FFFFFFF, $80000000, c0=1, c1=2, c2=4, c3=$100000 (c0 to c3 are user defined constants, c3 is for fp)
1_2, 0_1, $\langle$index$\rangle_2$= s$\langle$index$\rangle$p; stack index 0p-3p and discard [1]
1_2, 1_1, $\langle$index-4$\rangle_2$= s$\langle$index$\rangle$; stack index 4-7
1_1, $\langle$stack$\rangle_2$, $\langle$index$\rangle_2$= $\langle$stack$\rangle$s$\langle$index$\rangle$

T2:

0_2, $\langle$index$\rangle_3$= pin s$\langle$index$\rangle$; move TOS to s$\langle$index$\rangle$ and drop it.
else
pick T1= push selected stack item, coded as T1.

Abbreviations: pin s0 $\hat{=}$ drop, pin s1 $\hat{=}$ nip, pick s0 $\hat{=}$ dup, pick s1 $\hat{=}$ over, pick s1p $\hat{=}$ swap.

The multiplier issue doesn't push a result back, so it consumes the top of stack, too. "pass" allows to bypass the multiplier to make explicit use of the barrel shifter and double adder.

Rounded mul@ pushes one stack item (the upper half), unrounded pushes two items. The high order half of the used value is the TOS (big endian). Rounding is done after

[1] An ALU op will use TOS as first operand, the specified stack item as second operand and shift all stack items below the indexed operand. The result is written back to TOS, so never discarded. E.g. add s1p gives (.. a b c −− .. a b+c) and is the traditional stack add, adding and consuming the two topmost stack entries, and pushing the result back to the stack. s#p used with pick is a stack rotation; pick s1p reads stack item one, discards it and pushes it as a new TOS, thus swaps the two topmost stack entries (the effect of the FORTH word SWAP). pick s2p is equal to the FORTH word ROT.

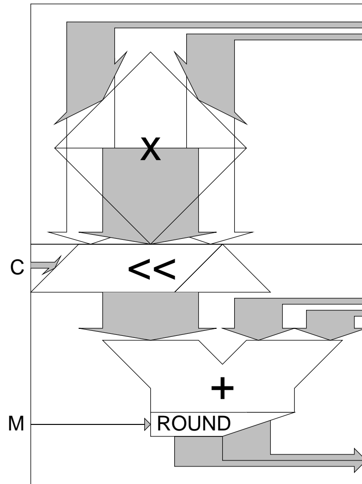

Figure 2: Multiply, shift and add unit

accumulation. Simply round fractional multiplication will divide the result by two; that's desired in block floating point operations as the FFT above.

T3:

0_1, $\langle\text{add}\rangle_1$, $\langle\text{negate}\rangle_1$, $\langle\text{round}\rangle_1$, $\langle\text{shift}\rangle_1$= [−]mul[r][<]@[+]:
 add : 0: no; 1: add dword from TOS:NOS
 negate : 0: no; 1: yes
 round : 0: no; 1: round by mode (IEEE round)
 shift : 0: no; 1: shift by count
1_1, $\langle\text{flag}\rangle_4$= compute flag

flag:									
	0-7 :	`t`	`0=`	`0<`	`ov`	`u<`	`u>`	`<`	`>`
	8-F :	`f`	`0<>`	`0>=`	`no`	`u>=`	`u<=`	`>=`	`<=`

The compare flags need a sub or subr before. All flag computation replace the TOS with the flag.

T4:

$\langle\text{group}\rangle_2$, $\langle\text{item}\rangle_3$=

		item							
		0	1	2	3	4	5	6	7
group	0	asr	lsr	ror	rorc	asl	lsl	rol	rolc
	1	ff1	popc	lob	loh	extb	exth	hib	hih
	2	sp@	loops@	loope@	ip@	sr@	cm@	index@	flatch@
	3	sp!	loops!	loope!	ip!	sr!	cm!	index!	—

ff1 pushes the amount of zeros preceeding the most significant one of TOS. popc pushes the amount of bits set in TOS. lob shifts the most significant byte (unsigned) to the least significant byte, loh does so for the half word. extb and exth sign extend a byte/half word. hib and hih shift a byte/half word to the most significant part of TOS (fractional

representation). The ∼@ (read "∼-fetch") instructions read special registers and push the result to the current stack, the ∼! (read "∼-store") store TOS to the special registers and pop it.

There is no integer division, as it can be easily emulated with the fast MAC, a table lookup and a Newton or Goldschmidt iteration. For division by constants, a multiply (with the "reziprocal" value) and a shift is usually enough.

T5:

0_1, $\langle$floating point$\rangle_4$: see below
1_1, $\langle$bit manipulation, pixel, ...$\rangle_4$:

		item			
		0	1	2	3
group	0	bfu	bfs	—	—
	1	cc@	cc!	—	—
	2	px4	px8	pp4	pp8
	3	—	—	—	—

The bit manipulation operations (bfu and bfs) take a "bit field descriptor" from the top of stack. This isn't really an exact descriptor of the bit field, but allows to form many possible bit field operations with very few instructions. The bitfield descriptor consists of three bytes, from least significant to most: r, n, m.

The bit field operation rotates the NOS r bits to the left, creates a n bit mask and rotates the mask m bits to the left. Then it ands the mask with the rotated NOS and leaves that in TOS, dropping NOS. bfs extends the sign bit (the MSB of the bit field). By using a mask rotation number $m > n$ and $r = 0$ parts of a bit field can be deleted using bfu, so bfu can do extract from a bit field (unsigned), clear in a bit field and create a bit field form a value that was perhaps in another bit field before. bfs can be used to do signed extraction.

cc@ and cc! load and store carry count in the DSP unit. Carry count is a 16 bit sign extended number, that counts carries (+1) and borrows (−1).

The pixel extract and pixel pack operations convert a double number on the stack from 4/8 bits per pixel to 8/16 bits per pixel and vice versa. Each wider pixel uses the pixel in the TOS value as more significant half and the NOS value as less significant half. The px4 and pp4 operations can be used for BCD arithmetics, too.

Mode and count (cm) are parts of the state register of each stack.

Flags affected: As each operation leaves its result in the TOS, flags are set according to the TOS value and to the operation extra results (carry and overflow). There is no true nop that doesn't affect flags (nop is equal to "or #0"). However, only add/sub or shift instructions affect carry flag, so or #0 leaves the flag as it was after an add/sub. To clear carry, use add #0, to set use sub #0.

Unary operations: not $\hat{=}$ xor #−1; neg $\hat{=}$ subr #0, inc $\hat{=}$ sub #−1, dec $\hat{=}$ add #−1.

Immediate numbers:

2_2, $\langle\text{const}\rangle_8$= 8 bit constant sign extended pushed on stack.
3_2, $\langle\text{const}\rangle_8$= 8 bit constant: shift TOS by 8 bit and insert constant into lower byte.

This allows to push four full 32 bit constants in four cycles. Most constants are shorter and an 8 bit signed constant is pushed in one cycle. E.g. the instruction sequence %10|\$12, %11|\$34, %11|\$56, %11|\$78 for one stack pushes the constant \$12345678 onto the stack (→ \$12 → \$1234 → \$123456 → \$12345678). It is not wise to use this for many constants, because parallel data moves allow four 32 bit constants to be pushed simultaneous without any delay. It's better to set up a data pointer to a constant field (per procedure) using ip and an immediate offset.

Data move operations:

If $\langle\text{move/modify}\rangle_1 = 1$:

$\langle\bar{\text{r}}/\text{w}\rangle_1$, $\langle\text{stack half}\rangle_1$, $\langle\text{dual move}\rangle_1$, $\langle\text{size}\rangle_2$, $\langle\text{mode}\rangle_3$, $\langle\text{reg}\rangle_2$= ld/st

else:

1_1, $\langle\text{stack half}\rangle_1$, $\langle\text{code}\rangle_3$, $\langle\text{mode2}\rangle_3$, $\langle\text{reg}\rangle_2$= ua/set/get/cache control & MMU/io/...
0_1, $\langle\text{shift}\rangle_1$, $\langle\text{constant}\rangle_8$= 8 bit sign extended, offset for the other side's data move op.

If $\langle\text{shift}\rangle$=1, the previous constant is shifted by 8 bits and the immediate constant is inserted into these lower 8 bits.

size: 0: byte "B", 1: half "H" , 2: word, 3: two words "2" or "F"
code: 0: add address, 1: subtract address, 2: get reg, 3: set reg, 4: cache & MMU control, 5: IO, 6–7: to be specified.

mode	syntax	address	R*n* becomes	comment
0	R*n*	R*n*+imm	R*n*	indirect
1	R*n* +N	R*n*+N*n*+imm	R*n*	indexed
2	R*n* N+	R*n*+imm	R*n*+N*n*	modify after
3	R*n* +N+	R*n*+N*n*+imm	R*n*+N*n*	modify before
4	s0b	s0+imm	R*n*	stack indirect b.e., reg=0;
	ipb	ip+imm	R*n*	ip relative b.e., reg=1;
	s0l	s0+imm	R*n*	stack indirect l.e., reg=2;
	ipl	ip+imm	R*n*	ip relative l.e., reg=3
5	R*n* +s0	R*n*+s0+imm	R*n*	stack indexed
6	R*n* s0+	R*n*+imm	R*n*+s0	stack modify after
7	R*n* +s0+	R*n*+s0+imm	R*n*+s0	stack modify before

The TOS for modes 4-7 is read at the begin of executing the instruction from the specified stack. However, the index is not consumed by the load/store.

Boundary crossing is detected if the outcoming sum of register and offset (either from the N*n* register or from s0) is greater or equal the L*n* limit register. In this case, the sum is replaced with R*n*, where the M lower bits are deleted (mask part of F*n*).

mode2: for update address: $\langle$modifier$\rangle_1$: 0=Nn (only if $\langle$stack$\rangle$), 1=s0, $\langle$destination reg$\rangle_2$. reg+modifier+immediate offset gives destination reg. Syntax Rdn= Rn [+|−(N|s0)].

Cache Control:

The instructions ccheck, cclr, cstore, cflush, cload, calloc and cxlock control the cache (mode2 from 0–3, the stack half bit is used to distinguish between cclr, cstore, cflush (bit clear) and cload, calloc, cxlock (bit set)). All these accesses use the address in Rn to select one cache line. The address part that is used to address the cache line set in usual accesses is used here, too. The lower bits select one line of the set. The higher bits are compared with the cache line's address; a mask count M specifies the lower bits of them not to compare. It is ensured, that a walk with constant offset through one page's address range will select all cache lines. Bit 0-1 specify which cache to check: data, instruction, higher stack's and lower stack's cache (even/odd is selected by the operation slot).

Example: an implementation may have 4K pages and 32 byte cache lines in a four way set associative cache. Thus, the lower 12 bit select the cache line. Therefrom the lower 5 bits select way and cache: 0 is way 0, 8 is way 1, $10 is way 2 and $18 is way 3, all in data cache.

ccheck returns the result of the check at the begin of the second instruction after its issue. Thus it has a delay slot as any other cache access, too. The result contains the status bits (MOESI for modified, owned, exclusive, shared, invalid) of the cache line, a flag whether the compare matched or failed and the number of unprocessed entries in the global write buffer. This helps to flush the cache without long interrupt latency.

A cache flush does not ensure that the cache is empty after the flush, but it ensures that cache and memory before the flush are consistent afterwards, thus all modified lines are written out and any valid data in the cache contains the same values as the corresponding memory. When interrupts are disabled, the cache is really empty, except those parts needed to run the cache flush loop.

MMU:

The 4stack's MMU is kept simple, but allows everything a modern OS needs. The MMU provides save kernel entries, protects pages form user space and kernel space from each other (if wanted), and defines different cache coherence requirements. Hardware implements only translation lookaside buffers (TLBs), the page table walk has to be done in software. This allows to use different page table formats, or to emulate a CPU that uses it's own virtual memory.

Each TLB contains a virtual address for four pages, beginning form a page number dividable by four. This allows to keep more pages in the MMU with the same amount of transistors, assuming a good locality among the used pages. There is a MMU for code pages, and a MMU for data pages.

The TLB update instruction takes a virtual/physical address pair. It selects the correct TLB slot by using either the lower bits of the virtual address (if not zero), or by searching an appropriate slot using the virtual address itself.

The lower part of a physical page address consists of two parts: access rights and page properties. Access rights are rwxrwx for user and supervisor (thus 6 bits). Properties consist of 3 bits for the system (cache coherence protocol, setmode) and 3 bits free for application purposes, which could be typically be used for access stamps (read, modified, executed), but these bits are not processed by the MMU.

The format thus is

	ac	su	usr
ccs	rwx	rwx	rwx

The setmode bit changes the execution mode when executing an instruction in this page, and the processor is in a mode not allowed to execute instructions.

The cache coherence protocol defineds four different types of pages: uncached, consistently updated, coherent cached and stale cached. This affects writes, and in case of “totally uncached”, reads, too. Writes to uncached pages are merged, but when they leave the write buffers, they are not stored in the cache. It is usually not recommended to use this mode, except you have a memory mapped device that takes rather large junks (up to 64 bytes) at once, e.g. a SCI device.

A page that is marked to be “consistently updated” is cached, but modifications are written out early (after they leave the inner write buffer, thus there are never dirty cache lines for this page). A possible application may be a frame buffer, but it is more recommended to flush frame buffer caches in the vertical blank instead of using this sort of write–through cache.

A “coherent cached” page uses a standard multiprocessor protocol for updated cache lines, thus writes to shared or unallocated cache lines generate a message on the bus. A “stale cached” page doesn’t generate these messages, it assumes that these pages are local (in fact, all reads are marked “with intent to modify”, so each other processor that holds this line has to invalidate it). Be careful when migrating processes that use “stale cached” pages! Cache lines that are completely generated by this processes may contain different (stale) contents in other CPUs.

Pages can be considered as empty, if none of the rwxrwx bits for supervisor and user are set. However, it is up to the TLB fault handler to convert (and validate!) page table entries.

I/O Control:

The operations out, outd, ins, ioq, inb, inh, in and ind control I/O ports. The *4stack* processor has an implementation dependent number of I/O read and write buffers. I/O accesses are not cacheable and not translated by the memory management unit; even if they go to memory addresses, and not to special port addresses. All I/O accesses are performed strictly sequential in an I/O queue. The access use the address in Rn plus the immediate constant scaled by 8. Each I/O read is a 64 bit read, each write is a 64 bit write to the corresponding address. On a 128 bit bus, the lowest valid bit in the address selects appropriate bus half.

The in start operation ins returns a input latch register number. The io query operation ioq returns true, if the passed register number is filled, false otherwise. An inb, inh, in or ind operation reads the port value and frees the latch register. The immediate offset allows to select the appropriate byte/half word/word out of the double word read. I/O accesses will cause a privilege violation exception when not processed in supervisor mode.

For get and set reg: 0=Rn, 1=Nn, 2=Ln, 3=Fn. The values 4-7 for mode are the same for 64 bit addresses (getd and setd).

Data operations either could use one stack or a dual operations for two stacks (controlled by the dual bit in the instruction). To indicate this, a different syntax for the stack location is used (e.g. 2&0: instead of 2:).

Dual operations place the low address half on the specified stack, the address+size half on the other stack. E.g. 2&0: as destination of the even stack data move unit means: The top half is pushed on stack 2, the lower half on stack 0; the stack bit contains 1.

All accesses should be size aligned. Double accesses can't be indexed with an immediate constant. Single accesses are always indexed with a constant, because the data move nop contains the index constant for the other operation.

Loads shall not be immediately followed by register fetches to the same stack, because the data unit uses one data path per stack and won't be able to push two values from two sources simultaneous.

Instruction control:

Stack 3 is used as instruction control stack. Calls push the return address on stack 3, return uses stack 3 to jump back with ip!. ip! changes the following instruction's next instruction address, thus the next instruction is executed before the indirect jump has an affect on the control flow (delay slot). Indirect calls have to use ip@ on stack 3 to save the return address. loops! and loope! have this delay slot, too. It takes another instruction until the fetch unit recognizes the changes.

Conditional branches read the flags of the specified stacks computed in the current instruction, and add the sign extended 11 bit offset to the IP of the next instruction to get the new address. If the ⟨copy/pop⟩-bit is true, the value is popped from the stack (condition prefix "?"), otherwise it remains (prefix ":"). The jump is performed, if all (and/or bit=1) or at least one (and/or bit=0) specified value mets the condition. If only one stack is specified, the and/or bit should be 0.

If no stack is selected as flag input and the combination mode is "or", the branch instruction is interpreted as setup for a loop. The flag and the copy/pop bit should be zero. A "do" initializes a loop by storing IP as loop start, IP+offset (sign extended) as loop end. The loop counter has to be initialized with index!. There is no need to unroll loops more than necessary to fill all instruction slots.

Loops are performed until the loop counter decrements to -1. Both pushing and restoring surrounding loops must be done with index@, loops@ and loope@ and the corresponding

~!s. The instruction fetch unit compares each new IP with loope, and if they are equal, it continues fetching at loops. This is not true for branch targets, because they are computed by the speculative fetcher, thus a branch to loope will leave the loop. A return to loope, however, will continue looping.

All conditional branches have a "likelihood" bit. This bit gives a hint to branch prediction logic, whether the branch will be taken (bit = 1), or not.

6 Conditional execution

Some simple algorithm don't use four integer units, and are conditional, too. However, even those can be executed in parallel with the aid of conditional execution of each per stack operation. I'll give a Z-buffer drawing algorithm as example (only the inner loop, drawing a horizontal line is given):

```
 xxx        xxx       xxx       xxx     ldh R1 :0&2    ldh R1 :1&3
 xxx        xxx       xxx       xxx     do loop
;c dz z' zr  c dz z' zr  c dz z' zr  c dz z' zr
 sub s1     sub s1    sub s1    sub s1  ?u< ?u< ?u< ?u<
;c dz z'   c dz z'   c dz z'   c dz z'
 pick s2    pick s2   pick s2   pick s2 st 0&2: R2 N+  st 1&3: R2 N+
 dup        dup       dup       dup     sth 0&2: R1 N+ sth 1&3: R1 N+
 nop        nop       nop       nop     :t :t :t :t
;c dz z    c dz z    c dz z    c dz z
 add s1     add s1    add s1    add s1  ldh 0&2: R1    ldh 1&3: R1
 nop        nop       nop       nop
loop:
```

To explain: The C source of this could be:

```
unsigned short *zbuffer;
color *screen;
color col;
unsigned short z,dz;

/* loop body */
for (...)
{
   if(z<*zbuffer)        /* sub s1  ?u< */
     {
        *screen=col;     /* pick s2  st 0&2: R2 N+ */
        *zbuffer=z;      /* dup      sth 0&2: R1 N+ */
```

```
    }
   screen++; zbuffer++;  /* included above    */
   z+=dz;                /* add s1      */
}
```

Thus all instructions execute conditionally, except the instruction to set and reset the condition. The ?⟨flag⟩ instruction takes the actual computed value from stack and deactivates the ALUs if false, the :⟨flag⟩ copies the actual computed value. :f reverts the activity of an ALU and is used for else parts. :t reactivates all ALUs. Once on an inactive ALU, all ?/:⟨flag⟩ won't change the state to active, nor change the stack's state, except :t and :f.

Parallel memory operations will perform their address updates, but won't perform stores, if the stack is deactivated, will read 0, if s0 is used as address update operator and won't push loaded values (but they will load, because they can't estimate the flag's state two cycles later).

With an possible clock rate of 100 MHz, we can draw 66 million Z-buffered pixel per second (peak rate, certainly). This would be enough to animate real time 3D pictures in HDTV quality. Shaded polygons would reduce the rate to 50 million pixels (with Gourand shading), replacing the last nop line by

```
pick s2     pick s2     pick s2     pick s2
add s4      add s4      add s4      add s4      ldh R1 :0&2   ldh R1 :1&3
pin s3      pin s3      pin s3      pin s3
```

and removing the ldh's in the previous instruction line.

The shading algorithm should care about color saturation, or a pixel add and saturate instruction is required. To compare: the MC88110 includes Z–buffering operations (pixel compare) which allows to draw two pixels in an average of 5.5 instructions, nearly 3 cycles per pixel then:

```
repl_both
  st.d   Colors,Framep,-8
repl_none
  ld.d   Zbufr,Zbufp,0
  padd   newZ,Zinc
  add    Framep,8
  add    Zbufp,8
  pcmp   CCR,newZ,Zbufr
  bb1    4,CCR,repl_both
  bb1    5,CCR,repl_none
  bb1    6,CCR,repl_first
  bb1    7,CCR,repl_secnd
...
```

Conditional execution can also be used to execute the end of an unrolled loop (1 to 4 iterations).

7 Floating point

		0	1	2	3
⟨floating point⟩	0	fadd	fsub	fmul	fnmul
	4	faddadd	faddsub	fmuladd	fmulsub
	8	fi2f	fni2f	fadd@	fmul@
	C	fs2d	fd2s	fxtract	fiscale

As integer multiply, floating point instructions are divided into two parts and visibly pipelined: into add and normalize, and multiply and normalize. Integer stacks are used as floating point stacks, and except fs2d, fd2s, fiscale, fxtract, fi2f and fni2f, all sources and results are 64 bit IEEE FP numbers. Each stack provides only one source number, and there is only one adder and one multiplier (to save space), which, however, can operate in parallel. There are bypasses for the adder (both from multiplier and adder) to allow a 3 cycle multiply and add, and an every cycle accumulation (the last a bit like i860).

Each source can be negated. Both adder and multiplier have two input latches, loaded by fadd, fsub, fmul and fnmul (negated mul). Odd stacks access one latch, even stacks the other; thus two fadds shall not be coded on two even stacks. The latches hold their input, even if only one fmul/fadd instruction is executed.

fiscale scales a FP exponent by an explicit integer argument, thus `fscale`$(m * 2^e, i) = m * 2^{e+i}$. The integer argument is in s0, the floating point number in s1 and s2. fxtract (f $--$ f n) inverts fiscale and produces a floating point number between 1 and 2 and the exponent integer.

fi2f and fni2f use subtract to convert integer to floating point and back. They change the MSB of the input integer, fi2f pushes $43300000, fni2f $C3300000, moves the result to the adder latch and starts an addition. Thus the difference of two converted integers is the difference of the two corresponding floating point values. Using 0 as second converted integer allows simple conversion. As the adder latch stays, block conversion does not need to setup the "zero" latch every cycle.

There is no fp division and square root (although these both are applyable in transistor logic) to keep the complexity in a reasonable range. Division and square root can be implemented using multiply and addition and this implementation helps to avoid problems with saving the FPU's state in some few cycles. Especially both add and multiply don't have special exceptions, as division by zero or square root of a negative number.

Examples: Vector multiplication:

```
 nop     nop    nop       nop       ldf 0: R1 N+   ldf 1: R1 N+
 nop     nop    nop       nop       ldf 0: R1 N+   ldf 1: R1 N+
 nop     nop    nop       nop       ldf 0: R1 N+   ldf 1: R1 N+
 fmul    fmul   nop       #n        ldf 0: R1 N+   ldf 1: R1 N+
 fmul    fmul   fmuladd   index!    do loop
 fmul    fmul   fmuladd   faddadd   ldf 0: R1 N+   ldf 1: R1 N+
loop:
 fmul    fmul   fmuladd   faddadd
 fmul    fmul   fmuladd   faddadd
 nop     nop    fmuladd   faddadd
 nop     nop    nop       fadd@     #0            stf 3: R2 N+
```

The pipeline in this example is, beginning with the data load, 5 cycles deep, but it needs at least 10 cycles to complete, because one load slot is occupied by the do instruction. Keeping one vector in cache, a memory interface with 128 bit bus size and 50 MHz will keep the pipeline filled.

Linpack ($\lambda x + A_i$):

```
;--       lambda    dest   --
 nop      nop       sp@      nop       ldf 0: R1      4 #
 pick 2s0 nop       drop     nop       ldf 0: R1 N+
 nop      nop       sp!      nop       ldf 0: R1 N+
 fmul     fmul      nop      #n        ldf 0: R1 N+
 fmul     fmuladd   fadd     index!    do loop
 fmul     fmuladd   fadd     fadd@     ldf 0: R1 +N+  stf 3: R1 N+
loop:
 nop      fmuladd   fadd     fadd@     0 #            stf 3: R1 N+
 drop     nop       pick 0s0 fadd@     0 #            stf 3: R1 N+
 nop      nop       sp!      nop
```

I use a stack as a "FIFO" — to get the second source in :-). This could be dangerous, sometimes… As source (A_i) and destination (likely in place: A_i) needs 128 bits per cycle (the cached vector x maybe not), a two cycle per step loop without this trick may be enough…

Block integer conversion (only inner loop):

```
 nop      #0       nop      #n
 fi2d     fni2d    nop      index!   do  loop
 fi2d     fadd@    nop      nop      ld 0: R1 N+  stf 1: R1 N+
loop:
 fi2d     fadd@    nop      nop      0 #          stf 1: R1 N+
 nop      fadd@    nop      nop      0 #          stf 1: R1 N+
```

Division can use NEWTON–RAPHSON's Iteration $x' = 2x - ax^2$ with $x_0 = -0.43a + 1.36$ (converts after four iterations and would take 29 cycles):

```
c0:  .fd -.43
c1:  .fd 1.36
; a         --        --        --
fdiv:
  fxtract   pick 0s0  loope@    loops@
  drop      and #min  pick 0s0  nop       .ip.f# c0  ldf 3: ipb
  fabs      nop       neg       nop       .ip.f# c1  ldf 1: ipb
  fmul      nop       nop       fmul
  fmuladd   fadd      flatch@   3 #
  index@    fadd@     nop       index!    do
  nop       fmul      fmul      nop
  fmul@     nop       flatch@   flatch@
  fmul      nop       nop       f2*
  fmulsub   nop       nop       fadd
  nop       fadd@     nop       nop
.loop
  index!    or s2p    pass      loops!
  nop       pick 2s0  drop      ret
  nop       fiscale   loope!
; --        1/a       --        --
```

A better way to calculate a/b is GOLDSCHMIDT's algorithm [1], because it makes better use of the independent multiplier and adder. Instead of taking 5 cycles per interation, GOLDSCHMIDT's algorithm takes only 3. A lookup table for the initial estimation improves both algorithms, and the latter then is able to divide two floating point numbers in 13 (1024 value table) or 16 (32 value table) cycles.

A variation of GOLDSCHMIDT's algorithm can be applied for sqare root (or reciprocal sqare root), taking 5 cycles per iteration and giving result after 22 (17 for larger table) cycles.

Abbreviations:

fabs=and #$7FFFFFFF
fneg=xor #$80000000
f2*=add c3
f2/=sub c3

8 Interrupts

The CPU has two states: normal program state and interrupt state. In interrupt state, the CPU reads four instructions out of the interrupt table (indexed by the interrupt in

service) and executes them. No other interrupt service is done until the interrupt code ends or forms a long interrupt. A long interrupt is formed by a jump in the interrupt code. All states have to be saved in the interrupt code and restored manually:

Sample long interrupt:

```
  sr@     ip@     sr@     sr@     stop@    stop@
  mul@    sr@     mul@    ip@
  flatch@ flatch@ flatch@ flatch@
  index@  loops@  loope@  jump longint

longint:
; perform some action, using do, mul, fadd and fmul
  index!  loops!  loope!  nop
  fadd    fadd    fmul    fmul
  pass    sr!     pass    ip!     restart  restart
  sr!     ip!     sr!     sr!
```

Each interrupt sets a bit in the interrupt mask and disables all other interrupts from this source until this bit is restored. The 8 interrupt mask bits in sr are global, thus 8 extern interrupts are distinguished. While in interrupt mode, the S bit is set, allowing to execute privileged instructions. Only interrupts and exceptions set the S bit, once cleared, sr! only affects the lower half of the real sr, and stores the top half in a special user status register, where only the trace–bit has an effect in user mode.

The indirect jump delay slot (after ip!) requires to fetch two instruction pointers, therefore the first invocation of ip@ in interrupt mode has a different meaning: it fetches the address of the second instruction, thus if the most recently executed instruction contained an ip!, it fetches this address. Because the corresponding ip! has to be the last instruction, it uses bit 2 to store the X flag of the status register, and clears the X flag of the status register. This allows to execute sr! on this stack one cycle before, thus the instruction space addressing occurs in the correct mode, too.

Interrupt mode uses a different set of memory registers. An interrupt stops the second stage of a load. The stopped address and the loading mode can be obtained with the opcode stop@ and restarted with restart. This is only necessary in long interrupts (however, it is obligatory then).

9 Exceptions

Besides interrupts, which are indeterministic and asynchronous, there are exceptions because of faulty instructions; e. g. a protection violation. The exception strategy is either to emulate these instructions, to restart them or to abort the program.

Number	Reason for exception	Unit that causes the fault
0–F	General exceptions	Interrupts and instruction faults
0	Reset	Reset pin low, double fault
1	Trace	Trace bit set
2	Instruction Fault	Invalid instruction
3	Privilege Violation	Decoder
4	FPU exception	FPU
5	reserved	
6	Instruction TLB Miss	Prefetcher
7	Instruction Memory Protection Fault	Prefetcher
8–F	Interrupts	Interrupt pin n

Number	Reason for exception	Unit that causes the fault
10–1F	Memory first stage faults	Memory unit
10	no exception	
11	Boundary crossing odd half	Memory unit
12	Misaligned Access odd half	Memory unit
13	Reserved	
14	Boundary crossing even half	Memory unit
⋮		
1A	Misaligned Access both halfs	Memory unit

Number	Reason for exception	Unit that causes the fault
20–2F	Memory second stage faults	PMMU
20	no exception	
21	TLB Miss odd half	PMMU
22	Memory Protection Violation	PMMU
24	TLB Miss even half	PMMU
⋮		
2A	MPV both halfs	PMMU

Number	Reason for exception	Unit that causes the fault
30–3F	Stack TLB Miss	Stack PMMU
31	Stack 3 TLB Miss	Stack 3's PMMU
32	Stack 2 TLB Miss	Stack 2's PMMU
⋮		
3F	All Stack's TLB Miss	All Stack's PMMU

The four stack PMMUs can cause four independent faults simultaneous, so theses faults are combined into a bit vector to form an exception vector.

Memory unit or PMMU may cause 2 independent faults simultaneous. These faults are combined to form an exception vector. Each exception vector must handle both faults. Only conflicting fault sources on the same memory unit will be sorted by priority. The

priority ranks from TLB miss (highest) to misaligned access (lowest). There is no possiblility for a memory protection violation to arise when a TLB miss had occured, and the misaligned memory access can be caught, wenn the TLB miss handler restarts the operation.

For some cases it is wise to serialize the errors; e. g. the OS may only handle only one outstanding page request. But the software TLB lookup may be able to access two page table entries at once, so serialization is only necessary, if both pages where found non–present.

The misaligned access fault does not occur on implementations with misaligned access supported except if an access crosses a page boundary and causes a TLB miss. The ld2nd in the exception causing memory unit will then cause an TLB miss fault and the restart will correctly load the misaligned value.

There is no guaranteed order between non–simultaneous faults, except that more severe faults may completely hide other faults (eg. reset will hide any of the other faults caused before). Interrupts are usually served first.

Exceptions and interrupts use the same call mechanism: the first four instructions are fetched out of the exception table and the memory register set switches to the interrupt memory register set. These four instructions will be executed and are guaranteed not to be interrupted. If they cause an exception, the CPU state is lost, so be carefully! If your code isn't exception–free, form a long exception by calling the dangerous code part.

The exception table has the fixed memory location of the first 2048 bytes of phyiscal memory (addresses 0–$800). If your OS needs relocatible fault vectors, emulate them in the real exception code segment by using long exceptions and indirect calling (you can use one of the exception memory registers to point at the fault vector table).

10 Implementation Hints

This section will give some hints for a high performant implementation of the described processor. It addresses stack register file, floating point unit, pipelining and prefetching.

10.1 Stack Register File

Stack registers are one of the parts of the critical path. Therefore a carefully design has to be made. With pushes and pops stack registers changes their names or have to be spilled or refilled. There are a maximum of 4 pushs or pops in one cycle (pass and st2 or mul@ and ld2's result in the same cycle). As the four topmost elements are preferred (the other stacks have access to them), they must be kept in a four-ported register file. These four read ports are addressed by the four stack units and thus allow access without collisions.

Each element must have an additional r/w port to the spill/fill buffer. To get around wire–crossing, each stack element has one connection to one of four rows in the spill/fill buffer. So register renaming is used to compute the correct physical address in the register file from the stack element number.

To write up to four result values in one cycle to the four topmost stack elements, or to fill up to four values from the spill/fill buffer, each of the four per stack registers must have one individual write port and a crossbar network to select the correct input for this cell. The top of stack serves as ALU input latch, too, so a write to the designated TOS must be forwarded to the ALU.

The actually computed TOS (and NOS for double word stores) have to be forwarded to the data unit and to the branch logic (the branch logic requires only computed flags). An ALU operation that consumes its input has to compute the zero flag for the resulting TOS.

The stack roll operations (pick s0p to pick s3p) requires a special capability of the stack register file: to move a selected number of elements to the element below. The store TOS network handles the picked element, which is passed through the ALU and therefore not lost. Thus the 4 element register file has to be organized like a sort of shift register in the orthogonal direction.

The spill/fill buffer has to provide a high data throughput, too. The topmost four elements may be accessed by the corresponding stack ALU (both read and write access) and certainly by the filled/spilled elements. The read row address is always the same as a fill request to that column; but write row addresses differ by one (and there is the possibility that both ports are used for a write access!). Each row must certainly have a connection to the corresponding cache. It is a good advice to write lines late (before they are overwritten in the circular buffer) and to fill them early (once they are free).

If each stack uses an own stack cache, no more than those four buffer rows are necessary, but if the data cache is used, some additional buffer rows should be good.

10.2 Stack Address Translation

All interrupts and exceptions use the same stack spill pointer as user code. This seems to cause trouble when the reason for the exception was the propagation of a stack into a protected or not available page. Therefore each stack needs to have at least two precalculated (and thus valid) page translations: the current page (certainly) and either the next or the previous page. The next page isn't critical, because it's needed for fills. Thus if a fill cases a page fault, subsequent spills go to a valid page. The previous page is critical, thus it must be available, if the stack lacks a certain amount of free space. A good choise is to get the correct address if half of the page is filled. The next page's physical address is only necessary, if the stack is almost empty, thus if the next fill would cross the page boundary.

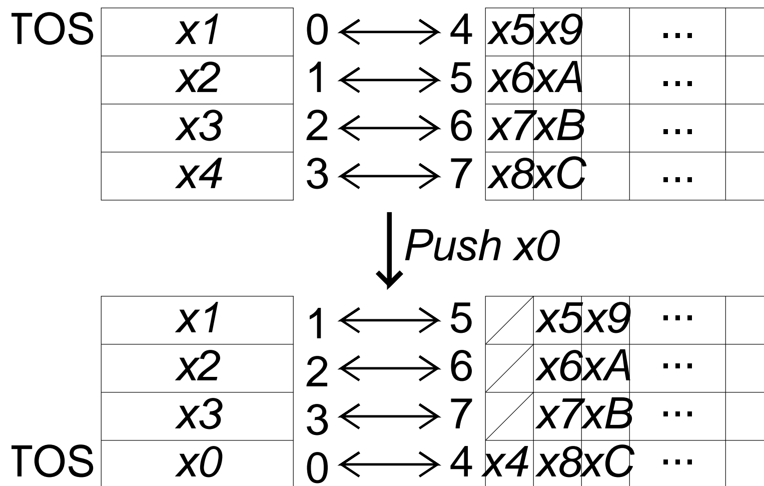

Figure 3: Stack register and stack buffer

10.3 Pipelining

High parallel instruction execution results in short basic blocks. While the cost of one conditional jump can be eliminated with speculative fetching, decoding (and maybe executing), the costs of consecutive conditional jumps increase exponentially, if the first condition isn't resolved for quite a long time.

The frequency of conditional jumps in usual programs is benchmarked at about 4 to 6 operations per jump, thus only one or two instructions on this processor. Even if conditional execution lengthens basic blocks in some important cases, conditional jumps are frequently and would destroy a constant pipeline stream in a long pipeline.

Therefore a short, three stage pipeline (fetch, decode, execute) must be a design goal. The fetch stage could be decoupled, thus prefetch sequential code as fast as the cache bandwidth is and wait, if the prefetch buffer is full. This saves power, as the cache and the instruction MMU are idle for some time. The following description uses a coupled fetch stage, because this is simpler to understand. If necessary, each stage is divided in two phases, the rising and falling clock edge.

Fetch: access the cache with the next instruction address and, if a MMU is present, computes the physical page address. On the second port, access the cache with the speculative next instruction address.

Compare the physical page address with the cache tag and select the correct cache line. For each cache way find out, if the instruction changes the control flow (bit 62 set) and compute the following instruction address. Forward the computed next instruction address to the speculative fetcher.

[Note: if a dual–ported i–cache and the parallel target address computation is too expensive, it is worth thinking to interleave normal and speculative fetching. This

will require a clock doubling and a speculation hit will give one half major cycle delay. Computing the branch target address then can be done in the first half of the decode stage.]

Decode: fill in nops for each unaddressed instruction unit. [Compute the branch target address, if not already done.]

Compute stack position to register number translations. Select data pathes for all read accesses. These data pathes will contain valid data at the beginning of the execute phase (plus an additional latch setup delay). Select the active units and forward the ALU opcodes. Compute the spill pathes.

Execute: start computation. Compute and select the write and refill pathes. All latches take only new data on rising clock edges. Compute the new renumbering of the stack elements. Compute memory address and access the data cache with the memory address of the previous cycle. Start MMU translation for data accesses.

Write results. The results need to be stable just before the rising clock edge. Compare physical page addresses and data cache tags. Select the correct cache element and forward it to the open write path (on reads). Store values from the store buffer to the accessed cache line (on writes). Move values to be stored to the store buffer (on rising clock edge).

[Note: if a dual ported data cache is too expensive, and if it is possible to reduce the address computation to a half major cycle, the two accesses may overlap, one using the second half of the execute phase to access cache and MMU, the other using the first half of the next cycle.]

10.4 Floating Point Unit

The floating point ISA uses a simplified model: compute and normalize. The real word looks different: For floating point multiplication, you need to setup the multiplier array ($m_1^T * m_2$); compute the diagonal sum of this array with carry save adding (3 to 2 adder or 7 to 3 adder); reduce the two results (sum and carries) with carry lookahead to one, round and shift left by one if the MSB is not set. The exponent sum is computed in parallel and incremented in the last step if the MSB was set.

For floating point adding, you have to compute the difference of the exponents and form the two's complements of the smaller, if sign bits are different. Then shift the smaller by the exponent difference to the right and add the inputs with carry lookahead. Count the leading zeros (leading ones if the result is negative; then build the two's complement of the result, too) and subtract the number from the bigger exponent. Shift the result into the appropriate position and round.

Using these four stages, the fmuladd and the faddadd instructions seems to have great problems: The fmuladd forwards a unfinished sum (a sum of two inputs) and the faddadd

forwards an unnormalized sum. Both is less critical than expected. The fmuladd can compute the sum while the shift amount is computed, the remaining 1 bit of possible difference has to be added to the input field length. The faddadd normalization is not necessary, as proper accuracy is maintained without it. Every step at most only one bit drops to the left; a wider internal field is sufficient to keep the result accurate. The one bit accuracy lost per step can be corrected in the first stage of the floating point add, before doing the shift. A clear MSB results in a shift by one to the left.

11 To be Done

Other potential useful instructions have to be found and specified. A supervisor/user model and appropriate protect strategies (if any) have to be found and specified.

A gcc port for the simulator... However, a machine description is likely not enough; register allocation on stack machines is completely different (see Phil Koopman's work); and a modified graph coloring algorithm for code scheduling is required.

References

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers; San Mareo, CA, 1990. Appendix A: Computer Arithmetic by David Goldberg.