A 16-bit instruction set

Niels Möller

1 Introduction

This file documents an attempt to define an instruction set with 16-bit op-codes and 16 general purpose registers. Current status: Most instructions are there, and they do fit in 16 bits. Some important features not yet specified. I also have a simulator and a primitive assembler.

Word size is $\ell = 64$ bits (variants with smaller native word size are possible).

1.1 Registers

There are 16 registers, R_0 to R_{15}. R_{15} is the program counter, and R_{14} is the link register for calls. R_{13} can be used as a stackpointer, but that's an ABI issue; the ISA and processor doesn't require any use of a stack. R_8 is the loop counter for the special branch-if-non-zero instruction.

1.2 About load and store

Loads and stores are big-endian. We always load and store full words. To make it easier to work with smaller quantities, unaligned effective addresses are allowed, with the following trick. A load with effective address p loads the word at $p \,\texttt{and}\, \neg 7$ (i.e., the low address bits are ignored for the actual memory access). But then the result is rotated depending on the low address bits: The word read is rotated left by $8\,(p \,\texttt{and}\, 7)$ bits. So the highest byte of the result is always the byte read from the given, possibly unaligned, address.

Stores do the inverse processing, the value to store is rotated right by $8\,(p \,\texttt{and}\, 7)$ bits and stored at $p \,\texttt{and}\, \neg 7$.

The most performance-critical loops are expected to always load and store full words anyway. Making access to partial words reasonably easy is intended to help for a common case outside of the most critical loops.

For load and store with index register, we use a trick suggested by Marcus Comstedt to encode an extra bit in the register ordering.

1.3 Constants

For most immediate values and offsets, we use 4 bits with the encoding in Table 1, and an explicit sign bit. The meaning of the sign bit depends a bit on the instruction, but in most cases it implies one's complement. Note that zero is not included. For operations where zero is a useful argument, special instructions are needed.

Code	Value	bits
0000	32	100000
0001	1	000001
0010	2	000010
0011	3	000011
0100	4	000100
0101	5	000101
0110	6	000110
0111	7	000111
1000	8	001000
1001	10	001010
1010	12	001100
1011	14	001110
1100	16	010000
1101	20	010100
1110	24	011000
1111	28	011100

Table 1: 4-bit encoding of immediate values.

This encoding is chosen to make for fairly simple hardware mapping from codes to values. To provide larger immediate values and offsets, adopt a suggestion by Leif Stensson. Use a prefix flag and a 60-bit prefix register, and new imm instruction including a constant, say 12 bits (could go down to 10 if needed). When this instruction is executed, the contents of the prefix register is shifted 12 bits left, and the 12 new bits are shifted in at the low end, and the prefix flag is set. The prefix register should be considered unsigned; sign bit is applied in the same way both with and without an active prefix register.

Instructions accepting an immediate value or offset check the prefix flag. If it is clear, the constant field is interpreted according to the above table. But if the prefix flag is set, the constant field (4 or 9 bits depending on instruction) is appended to the contents of the prefix register, and the low 64 bits are used as the immediate value or offset. The sign bit, if applicable, is applied to the resulting 64-bit value.

For arithmetic instructions and comparisons, the sign bit implies two's complement; to implement this, the addition unit(s) should have a carry input and a complement input, which the sign bit can be connected to. This seems to not work well for negative immediates to rsb (reverse subtract); for now, rsb is not supported. For logic operations, the sign bit implies one's complement.

The prefix flag is cleared when used, and it ought to be cleared after all branches (including using mov with the pc as destination). Maybe it's simplest to have it cleared by *all* instructions except imm.

Branches don't use this special coding. The instruction includes a 9 immediate bits and a sign bit. If the prefix flag is set, the 9 immediate bits are appended to the value in the prefix register. The offset is constructed by incrementing the immediate value, and then shifting it left one bit. Finally, this value is added or subtracted from the address of the next instruction, depending on the sign bit. If i denotes the unsigned immediate value (including prefix register if active), and o is the offset to add to the program counter, then we have $o = 2(i + 1)$ if

Code	Meaning
00	Not modified
01	Carry out
10	Signed not borrow
11	Signed overflow **(FIXME: Is this really needed?)**

Table 2: Options for setting the condition flag from an addition or subtraction.

Code	Mnemonic	Meaning
00	xshift	Shift in flag, flag unmodified.
01	xshiftc	Shift in flag, set flag from bit shifted out.
10	rshiftc	Shift in zero, set flag from bit shifted out.
11	ashiftc	Shift in sign bit, set flag from bit shifted out.

Table 3: Right shift with carry

sign bit is clear, and $o = 2\neg i$ if the sign bit is set, so we need to wire the sign to the complement input, and not sign to the carry input.

2 Conditional flag

There's only a single conditional flag, used for conditional jumps, conditional moves, and carry input to certain instructions. The flag can be set by add, sub, cmp, tst and xshift.

For addition and subtraction, using the flag as an input carry is optional. Subtraction is done as $a + \neg b + c$, so $c = 1$ means no borrow. When the flag is not used for carry input, carry in is zero for add and one for sub.

For flag output there are four possibilities, see Table 2. The overflow flag follows the ARM convention, including with carry input. The signed not borrow condition means that the the true sign of the signed result is non-negative. This makes the flag work as a signed greater-or-equal flag, and in addition, the result can be sign extended to register r using sub r, cc, r.

For shift right with carry (xshift), there are four variants, see Table 3. To do a signed (a+b)/2 as adds + xshift, we'd need to be able to shift in not carry. We introduce a not cc intruction for this and similar purposes.

There are some important loops where each iteration uses the value of the conditional flag produced by the previous iteration, e.g., a bignum add. To do that we need some branch instruction where the condition is based on the value of an ordinary register rather than the conditional flag. For this special case, we introduce a branch-if-non-zero instruction, hardwiring R_8 as the loop counter (a register number in the instruction reduces the number of bits available for the branch offset, and R_8 was chosen to not collide with floating point registers). There are some possible variants, like branch if non-negative, or decrement-and-branch-if-non-zero, but branch-if-non-zero seems to be the most generally useful. When using register R_8 as both loop counter and index register, one would often need to update it with some other constant than -1.

3 Shift instructions with shift count in a register

We only have one instruction for non-constant shifts, which can do all of left shift, right arithmetic shift, and right logical shift, depending on the shift count argument c.

If the sign bit is set, $c \geq 2^{\ell-1}$, we get an arithmetic right shift by $s = 2^{\ell} - c$ bits. If $s \geq 63$, the sign bit if r_d is copied to all bits of the destination register.

Otherwise, c is interpreted as an $\ell - 1$ bit two's complement number. If it is positive, i.e., bit $\ell - 2$ is zero, $c < 2^{\ell-2}$, we get a left shift by c bits, and if $c \geq 64$, the result is always zero. And if bit $\ell - 2$ is set, $c \geq 2^{\ell-2}$, we get a logical right shift by $s = 2^{\ell-1} - c$ bits, and if $s \geq 64$, we also get an always zero result.

I think it sounds more complicated than it is. To use this instruction, first compute c as a signed shift count, with positive meaning left shift. If any right shifting is intended to be arithmetic, we are done. Otherwise, clear the sign bit, which can be done with the fabs instruction.

We also have a two operand xshift, interpreting the shift count as above, but using the value of the condition flag for the first bit shifted in (if any; if the shift count is zero, the condition flag is unused). The encoding is stolen from shiftl with pc as one of the registers.

4 Multiplication

There are two multiplication instructions, mullo, returning the product mod 2^{ℓ}, and umulhi, returning the high half of an unsigned product. There are currently no immediate versions of these instructions. Several variants are missing: We have no signed variants of mulhi, and it's unclear if there are any usecases.

An earlier version of this document included an umull instruction, producing a fill 2ℓ bit product in two registers. This instructions was dropped after a discussion with Wesley W. Terpstra. The extra output port adds significant cost to the bypass-network in a super-scalar cpu. If we want another output port, to be able to get low and high half in the same cycle, it's better to add another, independent, multiplier unit; then one gets the flexibility of doing mulhi and mullo in parallel, or two mullo or two mulhi.

Additional input ports are not problematic in the same way. One possibly useful three-register instruction would be an "multiply and accumulate high", computing $\lfloor (a * b + c)/2^{\ell} \rfloor$.

5 Comparisons

Comparisons for equality is done using the cmpeq instruction. For inequality tests, there are more design options. Since the carry output from unsigned subtraction corresponds to not borrow, subc a, b sets the cc flag iff $a \geq b$. Therefore, the main unsigned compare instruction should be cmpugeq, setting the flag exactly like subc, but not storing the result of the subtraction. For consistency, the main signed comparison instruction is cmpsgeq. With signed not borrow defined as above, cmpsgeq sets the cc flag in the same way as subs.

We also define a tst a, b instruction, setting the cc flag if $a \, \texttt{and} \, b \neq 0$. This convention means that tst a, -2^k is equivalent to cmpugeq a, 2^k.

Redundant cmpugeq	Equivalent to	Encoding reused for
cmpugeq r, #-1	cmpeq r, #-1	cmpsgeq r, #0
cmpugeq r, #2	tst r, #¬1	cmpeq r, #0
cmpugeq r, #4	tst r, #¬3	cmpsgt r, # 8
cmpugeq r, #8	tst r, #¬7	cmpugt r, # 8

Table 4: Stolen immediate encodings for cmpugeq. These values are special only when no prefix is active.

Code	Meaning	bits
000	32	100000
001	10	001010
010	12	001100
011	14	001110
100	16	010000
101	20	010100
110	24	011000
111	28	011100

Table 5: 3-bit encoding of immediate values for cmpugt and cmpsgt.

Immediate comparisons need some special handling. We want to do immediate comparisons for equality, greater-or-equal and greater-than, with all 16 constants in Table 1, their negations, and zero. For signed and unsigned values. But, e.g., $x > 3$ is the same as $x \geq 4$, so we don't need all variants. And some comparisons can be done with the tst instruction, e.g., unsigned $x \geq 4$ is equivalent to $x \,\texttt{and}\, \neg 3 \neq 0$. We use three regular instructions, cmpeq, cmpugeq and cmpsgeq, using a sign bit, a 4-bit constant and any active prefix. With only a small tweak: When no prefix is active, some encodings for cmpugeq are stolen for other immediate comparisons. See Table 4.

The greater-than comparisons with small values, which aren't equivalent to some cmpgeq instruction, are then encoded as a special instruction using Table 5 to encode the desired operation.

6 Division

For integer division, we need a reciprocal instruction computing $\lfloor (2^{128}-1)/x \rfloor - 2^{64}$ for a normalized x, i.e., $2^{63} \leq x < 2^{64}$. Then with some extra book-keeping, we can get single-word unsigned division using umulhi, add, xshift, rshift. Unclear what the reciprocal instruction should do with unnormalized inputs, maybe we can have a two-operand instruction doing normalization and reciprocal at the same time, storing an appropriate shiftcount in a second destination operand?

7 Floating point

The first eight registers can be used for floating point operations. We also need some additional status register, not yet specified.

8 Exceptions and interrupts

We need a couple of different processor modes, identified by two bits in a system status register.

User mode: For normal execution of user programs.

Supervisor mode: Privileged mode, primarily entered by exceptions from user mode.

Supervisor exception mode: Entered by exception caused by supervisor mode. Often errors.

Interrupt mode: Entered as a result of an external interrupt signal.

Each mode gets its own copy of the system status register, the condition flag and prefix register (and any other status bits) and also its own copies of registers 12–15 (including the pc, the link register, which can also be used as scratch register, one register which can be used as a stack pointer, and one additional register). The last three modes are all privileged. They differ by having these separate registers, and in that exceptions can not be handled in supervisor exception mode or interrupt mode, and that interrupts cannot be handled in interrupt mode. If they occur nonetheless, they generate a reset exception.

For all exceptions, the link register (in the mode being switched to) gets some information about the source of the exception, with the low bits carrying the exception type. Since, at least for the interrupt mode, there are several possible previous modes, we need an additional two bits in the status register to identify which mode an rte instruction should return to. For each of the three types of exceptions, we have a separate exception vector register which is copied into the pc when the exception or interrupt occurs.

The following exceptions are needed.

Reset: Reset trap. Enters supervisor mode, with interrupts and mmu forced to disabled, and uses a fixed address rather than the exception vector register. On power on reset, all of the link register is zero; otherwise higher bits can indicate the reason for the reset.

System call trap: Invoked from user mode, target supervisor mode (maybe possible also from supervisor mode to supervisor exception mode). Arguments are passed in the regular registers, starting from register 0.

Unimplemented instruction: Caused by executing an unimplemented instruction.

Privileged instruction: Attempt at executing an privileged instruction in user mode.

TLB miss instruction: Generated when accessing a virtual address not present in the TLB cache. The virtual address (always 8-byte aligned) fits in the link register, provided that we need no more than 8 exception types.

TLB privileged access: Generated when attempting to access a virtual address which is present in the TLB, but fails the permission checks.

Code	Page size
aaa0	4 KB
a001	16 KB
0011	64 KB
0111	1 MB
1011	16 MB
1111	256 MB

Table 6: Page size coding

Interrupt: Hardware interrupt. Doesn't really need an exception type, since interrupts have their own exception vector. The link register can be set by the interrupt controller.

The system status register should also include a bit to disable interrupts, a bit to enable te MMU (or maybe it's easier to have it always enabled?) and an address-space id used by the Translation Lookaside Buffers (TLBs).

Access to system registers needs only two, privileged, instructions, rsys, and wsys. They use register r_1 to name the system register. rsys copies the value into r_0, while wsys copies the value from r_0.

(FIXME: What's needed for basic debugging? An explicit bkpt can use the same trap as system calls. For hardware watch points, we would either need a new exception and put it somewhere in the TLB lookup hardware. In principle maybe it could be done in software, by deleting the page from the TLB, and emulate memory accesses by code in supervisor mode. We could have a debug bit in the TLB which generates an exception on read or write accesses. May also need some trace bits for single instruction tracing.)

9 Memory management

Memory management is done mostly in software, with hardware only for the TLB (Translation Lookaside Buffer). This is a fully associative cache, which translates virtual addresses to physical addresses. Possible page sizes are 4 KB, 16 KB, 64 KB, 1MB, 16 MB, 256MB. Each entry in the TLB consists of two parts, the tag part containing the info needed to see if an access matches the entry, and the value part giving the physical address and other properties of the mapping.

For the tag, bits 11–14 encodes the page size, with a variable length code of at most 4 bits. Bits above the code are the start address of the page. See Table 6. **(FIXME: Review how the L4 microkernel represents mappings.)**

The low 11 bits includes a validity bit, maybe a shared bit, and an address space id of at least 8 bits. Some different ways of using the address space id are possible, but the simplest is that only TLB entries with address space which is either zero, or equal to the corresponding id in the system status register.

11	10	8	0
address/size	valid	?	asi

The value part encodes all information about the area being mapped. This includes the physical start address of the page, access permission bits, caching info (no cache, write-through cache, write-back cache), and io/strict order. Possibly also ARM-style sharing properties. There are plenty of bits; 1 petabyte of physical memory is 2^{38} pages of 4 KB, leaving 26 bits for other uses.

63	61	56	12	11	6	3	0
	cconfig		address	lock		u. access	p. access

In this layout, the empty fields are reserved. The cconfig field controls caching: 00 means no caching, 10 means cache with write-through, and 11 means cache with write-back. The special value 01 means no caching, and strict ordering of accesses (i.e., no reorder, no prefetch), suitable for memory mapped i/o registers.

The lock bit tells the least-recently-used hardware that this entry should never be a candidate for replacement. There are separate access bits for user mode and privileged mode, with three bits, rwx, for each.

The size of the address field implies that physical memory space is restricted to 2^{56} bytes, which ought to be sufficient. . . .

9.1 Bypassing the MMU

We can also let virtual addresses with the high bit set imply a one-to-one mapping to physical addresses, independent of the MMU. Using such addresses is allowed in privileged mode only. The low 56 bits give the physical address. Bits 61-62 are interpreted as cache control bits, reusing the layout of the TLB value word. With this feature, it is unclear if we need any status bit to disable the MMU; if we disable the MMU we need to either disable caching, or introduce some other mechanism to decide which areas can be cached, which seems messy.

10 Op-code allocation

code instruction

Load and store. Offsets are coded as c according to Table 1. Total of 0x5000 op codes (with some small holes). The instructions using an offset apply the prefix register, if active.

000s	n	c	d	ld	r_d, $[r_n, \#(-1)^s x]$	Load with offset (Table 1)
001s	n	c	d	st	r_d, $[r_n, \#(-1)^s x]$	Store with offset
0100	i	n	d	ld	r_d, $[r_n, r_i]$	Indexed load, $n < i$
0100	i	n	d	st	r_d, $[r_n, r_i]$	Indexed store, $n > i$

Besides load and store with indexed addressing, there's one additional instructions taking three registers, shiftl.

0101	b	c	d	shiftl	r_d, r_b, r_c

Shift r_d r_c bits, shifting in bits from r_b.

0101	1111	c	d	xshift	r_d, r_c

For long shift, r_d is unchanged if $r_c = 0$, and set to zero if $|r_c| \geq 64$. Otherwise, if $r_c > 0$, r_d is shifted left, shifting in bits from the high end of r_b, and if $r_c < 0$, r_d is shifted right, shifting in bits from the low end of r_b. Note that $b = c$ gives a rotate. For $b = 15$, cc is shifted in, not the pc.

Shift instructions, with 6-bit count ($c = 0$ is special).

0110	00cc	cccc	d	lshift	r_d, #c	Left shift
0110	0000	0000	d	clz	r_d	Count leading zeros
0110	01cc	cccc	d	rshift	r_d, #c	Logical right shift
0110	0100	0000	d	ctz	r_d	Count trailing zeros
0110	10cc	cccc	d	ashift	r_d, #c	Arithmetic right shift
0110	1000	0000	d	cls	r_d	Count sign bits
0110	11cc	cccc	d	rot	r_d, #c	Rotate left
0110	1100	0000	d	popc	r_d	Population count

0111	iiii	iiii	iiii	imm	#i	Prefix for constant/offset

Instructions with a (relatively) large offset to the pc. Offset is scaled by 2. All the instructions apply the prefix register, if active.

1000	00so	oooo	oooo	jmp	pc + $(-1)^s o$	Unconditional jump
1000	01so	oooo	oooo	jsr	pc + $(-1)^s o$	Subroutine call
1000	10so	oooo	oooo	bt	pc + $(-1)^s o$	Branch if true
1000	11so	oooo	oooo	bf	pc + $(-1)^s o$	Branch if false
1001	00so	oooo	oooo	bnz	pc + $(-1)^s o$	Branch if $r_8 \neq 0$
1001	01xx	xxxx	xxxx			Unassigned

Floating point operations.

1001	100a	aabb	bddd	fmac	r_d, r_a, r_b	"Fused" $d \leftarrow d + ab$
1001	1010	0sss	sddd	fldexp	r_d, r_s	Adds integer r_s to exponent.
1001	1010	10ss	sddd	fadd	r_d, r_s	
1001	1010	11ss	sddd	fsub	r_d, r_s	
1001	1011	00ss	sddd	fmul	r_d, r_s	
1001	1011	01ss	sddd	fdiv	r_d, r_s	
1001	1011	10ss	sddd	fcmpeq	r_d, r_s	Sets flag
1001	1011	11ss	sddd	fcmpgeq	r_d, r_s	Sets flag

1001	1100	00ss	sddd	fcmpgt	r_d, r_s	Sets flag

Single register floating point operations.

1001	1101	0000	0ddd	fs2d	r_d	Convert single to double.
1001	1101	0001	1ddd	fd2s	r_d	Convert double to single.
1001	1101	0010	0ddd	fui2d	r_d	Convert unsigned to double.
1001	1101	0010	1ddd	fd2ui	r_d	Convert double to unsigned.
1001	1101	0011	0ddd	fsi2d	r_d	Convert signed to double.
1001	1101	0011	1ddd	fsi2d	r_d	Convert double to signed.
1001	1101	0100	0ddd	fui2s	r_d	Convert unsigned to single.
1001	1101	0100	1ddd	fsi2s	r_d	Convert signed to single.

(Converting single precision to integer can go via double).

1001	1101	0101	0ddd	feqz	r_d	Set flag on $r_d = 0.0$
1001	1101	0101	1ddd	fgeqz	r_d	Set flag on $r_d \geq 0.0$
1001	1101	0110	0ddd	fgtz	r_d	Set flag on $r_d > 0.0$
1001	1101	0110	1ddd	fleqz	r_d	Set flag on $r_d \leq 0.0$
1001	1101	0111	0ddd	fltz	r_d	Set flag on $r_d < 0.0$

Instructions with 4-bit constant argument (see Table 1). Uses prefix register if active. oo field specifies carry output.

1010	0oos	cccc	d	add	r_d, $\#(-1)^s x$	$r_d \leftarrow r_d + (-1)^s x$
1010	1oos	cccc	d	add	r_d, cc, $\#(x \,\texttt{xor}\, -s)$	$r_d \leftarrow r_d + (x \,\texttt{xor}\, -s) + c$
1011	000s	cccc	d	mov	r_d, $\#(-1)^s x$	$r_d \leftarrow (-1)^s x$
1011	001s	cccc	d	and	r_d, $\#(x \,\texttt{xor}\, -s)$	$r_d \leftarrow r_d \,\texttt{and}(x \,\texttt{xor}\, -s)$
1011	010s	cccc	d	or	r_d, $\#(x \,\texttt{xor}\, -s)$	$r_d \leftarrow r_d \,\texttt{or}(x \,\texttt{xor}\, -s)$
1011	011s	cccc	d	xor	r_d, $\#(x \,\texttt{xor}\, -s)$	$r_d \leftarrow r_d \,\texttt{xor}\, x \,\texttt{xor}\, -s$
1011	100s	cccc	d	tst	r_d, $\#(x \,\texttt{xor}\, -s)$	Set flag on $r_d \,\texttt{and}(x \,\texttt{xor}\, -s) \neq 0$
1011	101s	cccc	d	cmpeq	r_d, $\#(-1)^s x$	Set flag on $r_d = (-1)^s x$
1011	110s	cccc	d	cmpugeq	r_d, $\#(-1)^s x$	Set flag on $r_d \geq (-1)^s x$ (unsigned)

except stolen cmpugeq encodings, see Table 4.

1011	111s	cccc	d	cmpsgeq	r_d, $\#(-1)^s x$	Set flag on $r_d \geq (-1)^s x$ (signed)

Immediate compares using the special encoding in Table 5. Doesn't accept any prefix.

1100	0000	sccc	d	cmpugt	r_d, $\#(-1)^s x$	Set flag on $r_d > (-1)^s x$ (unsigned)
1100	0001	sccc	d	cmpsgt	r_d, $\#(-1)^s x$	Set flag on $r_d > (-1)^s x$ (signed)
1100	001x	xxxx	xxxx			Unassigned (mullo?)
1100	01xx	xxxx	xxxx			Unassigned (mullo?)
1100	1xxx	xxxx	xxxx			Unassigned

Two-operand instructions

1101	0ioo	s	d	add	r_d, r_s	Add, carry in if $i = 1$, for oo, see Table 2
1101	1ioo	s	d	sub	r_d, r_s	Subtract, carry handling as above
1110	0000	s	d	mov	r_d, r_s	
1110	0001	s	d	movt	r_d, r_s	Move if flag set
1110	0010	s	d	movf	r_d, r_s	Move if flag clear
1110	0011	s	d	and	r_d, r_s	
1110	0100	s	d	or	r_d, r_s	
1110	0101	s	d	xor	r_d, r_s	
1110	0110	s	d	mullo	r_d, r_s	$r_d \leftarrow r_d r_s \bmod 2^\ell$
1110	0111	s	d	umulhi	r_d, r_s	$r_d \leftarrow \lfloor r_d r_s 2^{-\ell} \rfloor$

1110	1000	*s*	*d*	shift	r_d, r_s	See Sec 3 for meaning of r_s
(For rotate, use the shiftl instruction)						
1110	1001	*s*	*d*	injt8	r_d, r_s	Copy low r_s byte to high r_d byte
1110	1010	*s*	*d*	injt16	r_d, r_s	
1110	1011	*s*	*d*	injt32	r_d, r_s	
1110	1100	*s*	*d*	tst	r_d, r_s	Set flag on r_d `and` $r_s \neq 0$
1110	1101	*s*	*d*	cmpeq	r_d, r_s	Set flag on $r_d = r_s$
1110	1110	*s*	*d*	cmpugeq	r_d, r_s	Set flag on $r_d \geq r_s$ (unsigned)
1110	1111	*s*	*d*	cmpsgeq	r_d, r_s	Set flag on $r_d \geq r_s$ (signed)
1111	0000	*s*	*d*	ld	r_d, $[r_s]$	Plain load
1111	0001	*s*	*d*	st	r_d, $[r_s]$	Plain store
1111	001x	xxxx	xxxx			
1111	01xx	xxxx	xxxx			Unassigned 0x300

One-operand instructions.

1111	1000	00oo	*d*	add	r_d, cc, #0	$r_d \leftarrow r_d + c$
(FIXME: Used to be special sub cc, #8.)						
1111	1000	01oo	*d*	sub	r_d, cc, #0	$r_d \leftarrow r_d - 1 + c$
1111	1000	10mm	*d*	xshift	r_d	Single-bit right shift (Table 3).
1111	1001	0000	*d*	neg	r_d	
1111	1001	0001	*d*	not	r_d	$d \neq 15$
1111	1001	0001	1111	not	cc	
1111	1001	0010	*d*	bswap	r_d	Swap bytes
1111	1001	0011	*d*	recpr	r_d	Reciprocal
1111	1001	0100	*d*	jsr	r_d	Indirect subroutine call.
1111	1001	0101	*d*	fneg	r_d	Toggle sign bit
1111	1001	0110	*d*	fabs	r_d	Clear sign bit
1111	1001	0111	xxxx			
1111	1001	1xxx	xxxx			
1111	101x	xxxx	xxxx			Unassigned
1111	1100	xxxx	xxxx			System instructions
1111	1111	1111	1110	bkpt		Breakpoint
1111	1111	1111	1111	halt		Halt simulator

11 Remaining work

With the above op-code allocation, it looks liek we have plenty of opcode space left , can that really be correct? We have 3 blocks of 0x600 or more free opcodes. We could move instructions around a bit, putting the floating point ops earlier, and try to get space for some more imm ops as well as regular two-operand ops. Maybe we could even leave space for another branch instruction.

- System features: System call, interrupts, save and restore status flags and prefix register, load locked, store conditional, memory barrier, pre-fetch, MMU and TLB handling,.... Speaking of pre-fetch, there should be a way to clear a cache line so we can write to a memory block without first fetching the old contents.

- Missing immediate forms for multiplication instructions. Is this needed? And do we need signed mulhi?
- Hooks for SIMD unit or general co-processor.

Some nice-to-have features that have been left out:

- It would be nice with extract instructions (i.e., right shift by 56, 48 or 32 bits) with separate destination register.
- Similarly, it would be nice with clz and ctz with a separate destination register.
- A three-operand add is often useful to reduce the number of mov instructions. There's no space, but one might consider replacing the indexed load and store instructions. Or we could sacrifice a bit to get "alternate destination" for some instructions, storing the result into some fixed register, possibly r0.
- The immediate prefix instruction could be reduced from 12 to 10 bits, if we need additional instructions.