A 16-bit instruction set

Niels Möller

1 Introduction

This file documents an attempt to define an instruction set with 16-bit opcodes and 16 general purpose registers. Current status: Not extremely pretty, and some important features not yet specified. But at least the instructions do fit in 16 bits.

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. **(FIXME: Stack pointer?)**

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, allow unaligned effective addresses, with the following trick. A load at the effective address is p, loads the word at $p \,\texttt{and}\, -8$ (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 $8(p \,\texttt{and}\, 7)$ bits left. 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 $8(p \,\texttt{and}\, 7)$ bit right and stored at $p \,\texttt{and}\, -8$.

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 offset, optionally do post-increment (for offset > 0) and pre-decrement (for offset < 0).

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 or 5 bits with the encoding in Table 1 and Table 2. Note that zero is not included, for operations where zero is a useful argument, special instruction is needed.

To provide larger immediate values and offset, adopt a suggestion by Leif Stensson. Use a prefix flag and a prefix register (the latter 60 bits, so that we can copy both contents and a couple of flags into a 64-bit register for context switches and the like), and new imm instruction including a constant, say 12

Code	Value	Code	Value
0000	1	1000	-1
0001	3	1001	-3
0010	2	1010	-2
0011	6	1011	-6
0100	4	1100	-4
0101	12	1101	-12
0110	8	1110	-8
0111	16	1111	-16

Table 1: 4-bit encoding of immediate values.

Code	Value	Code	Value
00000	1	10000	-1
00001	3	10001	-3
00010	5	10010	-5
00011	7	10011	-7
00100	2	10100	-2
00101	6	10101	-6
00110	10	10110	-10
00111	14	10111	-14
01000	4	11000	-4
01001	12	11001	-12
01010	20	11010	-20
01011	28	11011	-28
01100	8	11100	-8
01101	16	11101	-16
01110	24	11110	-24
01111	32	11111	-32

Table 2: 5-bit encoding of immediate values.

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

Table 3: Options for setting the condition flag from an addition or subtraction.

Code	Meaning
00	Shift in flag, flag unmodified.
01	Shift in flag, set flag from bit shifted out.
10	Shift in zero, set flag from bit shifted out.
11	Shift in sign bit, set flag from bit shifted out.

Table 4: Right shift with carry

bits (could go down to 10 if needed). When this instruction is executed, if the flag is clear, the constant is sign extended and copied to the prefix register. If the flag is set, the contents of the prefix register is shifted 12 bits left, and the 12 new bits are shifted in at the low end.

Instructions accepting an immediate value or offset check the prefix flag. If it is clear, the constant field is interpreted according to the above tables. But if the prefix flag is set, the constant field (4, 5 or 10 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 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.

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, and tst.

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. (There's no sub with immediate, to do sub with carry and an immediate value, one must adjust the constant).

For flag output there are four possibilities, see Table 3. 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 4. **(FIXME: To do signed (a+b)/2 as adds + xshift, we'd need to be able to shift in not carry. Or introduce a "not cc" intruction.)**

Redundant cmpugeq	Equivalent to	Encoding reused for
cmpugeq r, #-1	cmpeq r, #-1	cmpsgeq r, #0
cmupgeq r, #1	tst r, #-1	cmpeq r, #0
cmupgeq r, #2	tst r, #-2	cmpugt r, # 8
cmupgeq r, #4	tst r, #-4	cmpsgt r, # 8
cmupgeq r, #8	tst r, #-8	unused
cmupgeq r, #16	tst r, #-16	unused
cmupgeq r, #32	tst r, #-32	unused

Table 5: Stolen immediate encodings for cmpugeq. These values are special only when no prefix is active.

Code	Meaning
0000	10
0001	12
0010	14
0011	16
0100	20
0101	24
0110	28
0111	32
1000	−10
1001	−12
1010	−14
1011	−16
1100	−20
1101	−24
1110	−28
1111	−32

Table 6: 5-bit encoding of immediate values for cmpugt and cmpsgt.

3 Comparisons

Comparisons for equality is done using the cmpeq instruction. **(FIXME: Or cmpneq instead? Then cmpneq #0 is equivalent to tst #-1. Or use one of the unused encodings for cmpeq #0.)**. 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 non 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.

Immediate comparisons need some special handling. We want to do immediate comparisons for equality, greater-or-equal and greater-than, with all 32 constants in Table 2, 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 $x \texttt{ and } -4 \neq 0$.

We use three regular instructions, cmpeq, cmpugeq and cmpsgeq, using 5-bit constants and any active prefix. With only a small tweak: When no prefix is active, some encodings for cmpuleq are stolen for other immediate comparisons. See Table 5.

The greater-than comparisons with small values, which aren't equivalent to some cmpgeq instruction, are then encoded as a special instruction using Table 6 to encode the desired operation.

4 Division

For integer division, we need a reciprocal instruction computing $\lfloor 2^{128}/x \rfloor - 2^{64}$ 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?

5 Floating point

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

6 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.

0000	c	n	d	ld	r_d, $[r_n, o]$	Load with offset (Table 1)
0001	c	n	d	ld	r_d, $[r_n, o]!$	Load with offset, update r_n
0010	c	n	d	st	r_d, $[r_n, o]$	Store with offset
0011	c	n	d	st	r_d, $[r_n, o]!$	Store with offset, update r_n
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 are two additional instructions taking three registers, umull and shiftl.

0101	h	a	d	umull	r_d, r_h, r_a	$\langle r_h, r_d \rangle \leftarrow r_a r_d$
0110	b	c	d	shiftl	r_d, r_b, r_c	Shift r_d r_c bits, shifting in bits from r_b.

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.

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

Instructions with 5-bit constant argument (see Table 2). Uses prefix register if active.

1000	iooc	cccc	d	add	r_d, #x	i and oo specify carry use
1001	iooc	cccc	d	rsb	r_d, #x	$r_d \leftarrow \#x - r_d$ (with carry)
1010	000c	cccc	d	mov	r_d, #x	
1010	001c	cccc	d	and	r_d, #x	
1010	010c	cccc	d	or	r_d, #x	
1010	011c	cccc	d	xor	r_d, #x	
1010	100c	cccc	d	tst	r_d, #x	Set flag on r_d `and` $x \neq 0$
1010	101c	cccc	d	cmpeq	r_d, #x	Set flag on $r_d = x$
1010	110c	cccc	d	cmpugeq	r_d, #x	Set flag on $r_d \geq x$ (unsigned)

except stolen cmpugeq encodings, see Table 5.

1010	111c	cccc	d	cmpsgeq	r_d, #x	Set flag on $r_d \geq x$ (signed)

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

1011	00cc	cccc	d	lshift	r_d, #c	Left shift
1011	0000	0000	d	clz	r_d	Count leading zeros
1011	01cc	cccc	d	rshift	r_d, #c	Logical right shift
1011	0100	0000	d	ctz	r_d	Count trailing zeros
1011	10cc	cccc	d	ashift	r_d, #c	Arithmetic right shift
1011	1000	0000	d	cls	r_d	Count sign bits
1011	11cc	cccc	d	rot	r_d, #c	Rotate left
1011	1100	0000	d	popc	r_d	Population count

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

1100	00oo	oooo	oooo	jmp	pc + o	Unconditional jump

1100	01oo	oooo	oooo	jsr	pc + o	Subroutine call
1100	10oo	oooo	oooo	bt	pc + o	Branch if true
1100	11oo	oooo	oooo	bf	pc + o	Branch if false

Two-operand instructions

1101	0ioo	s	d	add	r_d, r_s	Add, carry in if $i = 1$, for oo, see Table 3
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	lshift	r_d, r_s	$r_s > 0$ means left
1110	1001	s	d	ashift	r_d, r_s	$r_s > 0$ means left
(For rotate, use the shiftl instruction)						
1110	1010	s	d	injt8	r_d, r_s	Copy low r_s byte to high r_d byte
1110	1011	s	d	injt16	r_d, r_s	
1110	1100	s	d	injt32	r_d, r_s	
1110	1101	s	d	tst	r_d, r_s	Set flag on r_d `and` $r_s \neq 0$
1110	1110	s	d	cmpeq	r_d, r_s	Set flag on $r_d = r_s$
1110	1111	s	d	cmpugeq	r_d, r_s	Set flag on $r_d \geq r_s$ (unsigned)
1111	0000	s	d	cmpsgeq	r_d, r_s	Set flag on $r_d \geq r_s$ (signed)
1111	0001	s	d	ld	r_d, $[r_s]$	Plain load
1111	0010	s	d	st	r_d, $[r_s]$	Plain store
1111	0011	xxxx	xxxx			Unassigned (0x100)

Immediate compares using the special encoding in Table 6. Doesn't accept any prefix.

1111	0100	cccc	d	cmpugt	r_d, #x	Set flag on $r_d > x$ (unsigned)
1111	0101	cccc	d	cmpsgt	r_d, #x	Set flag on $r_d > x$ (signed)

One-operand instructions.

1111	0110	00mm	d	xshift	r_d	Single-bit right shift (Table 4)
1111	0110	01oo	d	adc	r_d, #0	Add carry, flag output according to Table 3
1111	0110	1000	d	neg	r_d	
1111	0110	1001	d	bswap	r_d	Swap bytes
1111	0110	1010	d	jsr	r_d	Indirect subroutine call.
1111	0110	1011	d	recpr	r_d	Reciprocal
Unassigned area (0x140)						

Floating point operations.

1111	100a	aabb	bddd	fmac	r_d, r_a, r_b	"Fused" $d \leftarrow d + ab$
1111	1010	0sss	sddd	fldexp	rd, r_s	Adds integer r_s to exponent.
1111	1010	10ss	sddd	fadd	r_d, r_s	
1111	1010	11ss	sddd	fsub	r_d, r_s	
1111	1011	00ss	sddd	fmul	r_d, r_s	
1111	1011	01ss	sddd	fdiv	r_d, r_s	

1111	1011	10ss	sddd	fcmpeq	r_d, r_s	Sets flag
1111	1011	11ss	sddd	fcmpleq	r_d, r_s	Sets flag
1111	1100	00ss	sddd	fcmplt	r_d, r_s	Sets flag
Single register floating point operations.						
1111	1100	0100	0ddd	fneg	r_d	
1111	1100	0100	1ddd	fabs	r_d	
1111	1100	0101	0ddd	fs2d	r_d	Convert single to double.
1111	1100	0101	1ddd	fd2s	r_d	Convert double to single.
1111	1100	0110	0ddd	fui2d	r_d	Convert unsigned to double.
1111	1100	0110	1ddd	fd2ui	r_d	Convert double to unsigned.
1111	1100	0111	0ddd	fsi2d	r_d	Convert signed to double.
1111	1100	0111	1ddd	fsi2d	r_d	Convert double to signed.
1111	1100	1000	0ddd	fui2s	r_d	Convert unsigned to single.
1111	1100	1000	1ddd	fsi2s	r_d	Convert signed to single.
(Converting single precision to integer can go via double).						
1111	1100	1001	0ddd	feqz	r_d	Set flag on $rd = 0.0$
1111	1100	1001	1ddd	fgeqz	r_d	Set flag on $rd \geq 0.0$
1111	1100	1010	0ddd	fgtz	r_d	Set flag on $rd > 0.0$
1111	1100	1010	1ddd	fleqz	r_d	Set flag on $rd \leq 0.0$
1111	1100	1011	0ddd	fltz	r_d	Set flag on $rd < 0.0$
1111	1100	1011	1ddd			Unassigned area(0x148)
1111	111x	xxxx	xxxx			Reserved (0x200).

7 Remaining work

With the above op-code allocation, we have 0x488 op-codes left, out of the space of 0x10000. This is sufficient for two single two-operand instructions or a dozen of of single-operand instructions. Important missing pieces, roughly in order of decreasing importance:

- System features: System call, interrupts, save and restore status flags and prefix register, atomic operations, memory barrier, pre-fetch, ... 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. Maybe we have to sacrifice independent destination arguments to umull to make room? Instead, store the result in an adjacent pair of registers.
- Hooks for SIMD unit or general co-processor.

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

- Signed long multiplication (smull and smulhi). Could add smulhi, but there's no space for a three-operand smull.
- 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.
- The fabs and fneg instructions are trivial bit operations, a bit too trivial to have their own instructions. Would be nice if they could be generalized.
- 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.