CS201: Architecture and Assembly Language

Lecture Three

Brendan Burns

Arithmetic for computers

Previously we saw how we could represent unsigned numbers in binary and how binary addition is performed. Today we’ll flesh out many of the details.

- Representing Negative Numbers
- Subtraction, Multiplication and Division

A reminder

First a quick reminder on binary, octal and hexidecimal.

Binary	Octal	Hex	Decimal	Binary	Octal	Hex	Decimal
0000	0	0	0	1000	10	8	8
0001	1	1	1	1001	11	9	9
0010	2	2	2	1010	12	A	10
0011	3	3	3	1011	13	B	11
0100	4	4	4	1100	14	C	12
0101	5	5	5	1101	15	D	13
0110	6	6	6	1110	16	E	14
0111	7	7	7	1111	17	F	15

Signed Numbers

We've already seen how to represent unsigned numbers. How could we do that?

- We'll we could just add a sign bit. . .
- But that's problematic:
  - Which side?
  - Negative Zero? (Univac had it...)
  - It slows down adding.
  - Can we solve these? Sure!

Two’s Compliment

In the end, since no representation was really pretty, hardware designers chose the one which made the hardware easiest:

$$\begin{aligned}
00000000000000000000000000000000_{two} &= 0_{ten} \\
00000000000000000000000000000001_{two} &= 1_{ten} \\
&\dots \\
01111111111111111111111111111110_{two} &= 2,147,483,646_{ten} \\
01111111111111111111111111111111_{two} &= 2,147,483,647_{ten} \\
10000000000000000000000000000000_{two} &= -2,147,483,648_{ten} \\
10000000000000000000000000000001_{two} &= -2,147,483,647_{ten} \\
&\dots \\
11111111111111111111111111111110_{two} &= -2_{ten} \\
11111111111111111111111111111111_{two} &= -1_{ten}
\end{aligned}$$

Two’s Compliment Cont.

This is known as *Two’s Compliment Notation*

- It does have the problem that there is one negative number without a matching positive number
- Also adding 1 to 2,147,483,647 gives you -2,147,483,648
- But it has a sign bit (the most-significant), and all numbers are computed in the same way

$$(x31 \times -2^{31} + x30 \times 2^{30} + \ldots + x1 \times 2^{1} + x0 \times 2^{0})$$

Two’s Compliment (Cont.)

If you’re loading in a byte or half-word, how its loaded varies depending on whether its signed or not.

- `ldrsh r1, [r2]` loads a signed half-word
- `ldrsb r1, [r2]` loads a signed byte
- In order to do this, the load performs sign-bit extension
- $11111110_{two} \rightarrow 11111111111111111111111111111110_{two}$
- The same is performed for signed half-words.

Two’s Compliment (Cont.)

Two’s compliment is also convenient because negating a number is as simple as inverting the number and adding one to it:

$$\begin{aligned}
-1 \times 7 &= NOT(0111_{two}) + 0001_{two} \\
-1 \times 7 &= 1000_{two} + 0001_{two} \\
-1 \times 7 &= 1001_{two} \\
-1 \times 7 &= -7_{ten}
\end{aligned}$$

Binary Addition

Previously we’ve seen how to do binary addition, for example:

- $15 + 23$
- What’s cool is that it works for negatives too!
- $12 + -10$

Binary Subtraction

Binary subtraction proceeds pretty much exactly like addition:

- $10 - 3$
- Alternatively we can negate the number and add.
- $10 - 3 = 10 + -3$

Binary Multiplication

It turns out that binary multiplication is exactly what you'd expect too:

- 12 * 13
- But we need to worry overflowing of our value.
- Fortunately ARM comes with a special multiply instruction
- `U,SMULLL r0,r1,r2,r3` unsigned/signed multiplies `r2 * r3` and stores the result into both `r0` lo-word and `r1` hi-word.

Multiplication Algorithms

So how does this actually work?

1. Set the product to multiplier
2. Is the l.s.b. of the product 0? If so, goto 3
3. Add the multiplicand to the product
4. Shift the product right 1 bit
5. Repeat 32 times

Signed Multiplication

Signed multiplication is trickier. An efficient algorithm for performing this is Booth’s algorithm:

1. Set the product to the multiplier
2. If the last two bits are 00 or 11 goto 5
3. If the last two bits are 01 add the multiplicand to the left half of the pduct
4. If the last two bits are 10 subtract the multiplicand from the left half od the product
5. Shift the product right by one

Booth's Example

- $4_{ten} \times -6_{ten} = -24_{ten}$

Why does this work?

- Consider $a \times b$
- At each step in Booth's algorithm we evaluate $(a_{i-1} - a_i)$, if its zero we do nothing, if its 1 we add b, if its -1 we subtract b.
- Thus at the end of the algorithm we have:

$$(a_{-1} - a_0) \times b \times 2^0 \ldots (a_{30} - a_{31}) \times b \times 2^31$$

- By factoring out b we get $b \times a_0 \times 2^0 \ldots a_{31} \times 2^31$ or $a \times b$.

Binary Division

Binary division proceeds much like binary multiplication

- $1010101/110$

An algorithm for division

The division algorithm is much like multiplication

1. Remainder gets the quotient
2. Shift the remainder to the left 1 bit
3. Subtract the divisor from the left half of the remainder
4. If remainder is < 0 add the divisor to the left half of the remainder and shift the remainder left with a 0 rightmost bit
5. If the remainder ≥ 0 shift the remainder to the left with a 1 rightmost bit
6. repeat 32 times

Signed Division

It turns out there's no cool algorithm for signed division. The best is to treat them as unsigned and set the sign bit afterwards. The tricky thing is to set the remainder's sign.

- We do this by noting that if the quotient should be negative and finding the remainder which works.

```
 7 /  2 =  3 r 1
 7 / -2 = -3 r +1
-7 /  2 = -3 r -1
-7 / -2 =  3 r -1
```

Dividing in Assembly

Of course the ARM chip doesn’t have a divide operator, so on the ARM you actually have to manually do out the division in assembly language:

- You can simply implement the algorithm described above in assembly.
- Or you can use a simpler algorithm (like repeated subtractions)
- In either case you’ll need *branch* instructions
- *Branch* instructions change the location of the program counter. (we’ll see them soon)

Condition Flags

Previously I mentioned that the CPU had a *status* register.

- It keeps track of the status of various things
- Two of them are:
  - The carry flag
  - The overflow flag
- Their necessary to determine the relationship between the *true* result and the actual result.

The Carry Flag

The carry flag indicates if there has been a carry out of the bounds of the result.
This only happens when the actual result is not equal to the true result

$$\begin{array}{ccccc} & 0 & 0 & 1 & 0 \\ + & 1 & 0 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 \\ & C & = & 0 & \end{array} \qquad \begin{array}{ccccc} & 0 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 \\ & C & = & 0 & \end{array} \qquad \begin{array}{ccccc} & 1 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 1 \\ \hline & 0 & 0 & 0 & 1 \\ & C & = & 1 & \end{array}$$

The Overflow Flag

Well, that worked for unsigned numbers, what about signed ones?

- The ’V’ flag in the status register indicates oVerflow.
- Overflow indicates that we have run out of bits to store our result
- fortunately we can reconstruct it from the flag.

The overflow flag

If the signs don't match, we're guaranteed not to have overflow,(why?) so the overflow flag is never set. If the signs are the same, there are four cases:

$$\begin{array}{ccccc} & & & & \\ & 0 & x_1 & x_2 & x_3 \\ + & 0 & y_1 & y_2 & y_3 \\ \hline & 0 & r_1 & r_2 & r_3 \\ & V & = & 0 & \end{array} \qquad \begin{array}{ccccc} & 1 & & & \\ & 0 & x_1 & x_2 & x_3 \\ + & 0 & y_1 & y_2 & y_3 \\ \hline & 0 & r_1 & r_2 & r_3 \\ & V & = & 1 & \end{array}$$

Overflow, Cont.

$$\begin{array}{ccccc} 1 & & & & \\ & 1 & x_1 & x_2 & x_3 \\ + & 1 & y_1 & y_2 & y_3 \\ \hline & 0 & r_1 & r_2 & r_3 \\ & V & = & 1 & \end{array} \qquad \begin{array}{ccccc} 1 & 1 & & & \\ & 1 & x_1 & x_2 & x_3 \\ + & 1 & y_1 & y_2 & y_3 \\ \hline & 1 & r_1 & r_2 & r_3 \\ & V & = & 0 & \end{array}$$

Subtraction?

What about in the case of subtraction?

- We've already handled signed subtraction (its just addition)
- For unsigned subtraction, the meaning of C is reversed.
- That is if $C = 1$ the actual result = the true result
- If $C = 0$ the actual result neq the true result
- This is because subtraction is really inverted addition

Other flags

The status register has two other flags that are also set by addition/subtraction operations (and multiplication if requested)

- The Z register is set to 1 if the result of the operation is zero
- The N register is set to 1 if the hi-bit of the result is one. (e.g. this number is negative *if in signed representation!*)

Condition Flag Summary

Flag	Unsigned meaning	Signed Meaning
N = 0	No meaning	Actual $\geq$ 0
N = 1	No meaning	Actual < 0
Z=0	Actual $\neq 0$	Actual $\neq$ 0
Z=1	Actual = 0	Actual = 0
C=0	Actual = True if ADD	No Meaning
	Actual $\neq$ True if SUB	No Meaning
C=1	Actual $\neq$ True if ADD	No Meaning
	Actual =True if SUB	No Meaning
V=0	No Meaning	Actual = True
V=1	No Meaning	Actual $\neq$ True