Arithmetic operations in assembly language

Prof. Gustavo Alonso
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
http://www.inf.ethz.ch/department/IS/iks/

Binary

- You are probably familiar with decimal and hexadecimal systems. Internally, a CPU works only with binary (0 or 1).
- With N bits, one can code \(2^N\) different states.
- Why binary, hexadecimal, octal ... ?
- To represent a decimal character, we need 4 bits (0, 1, ..., 9 = 10 different states) but not all possible combinations are used, i.e., in binary, decimal cannot be efficiently represented.
- For hexadecimal characters, we also need 4 bits (0, 1, 2, ..., E, F = 16 different states) and all combinations are used (the representation is space efficient).
- For octal, we need 3 bits (0, 1, ..., 6, 7 = 8 states) and the representation is compact. But we cannot efficiently store octal in bytes and words.

Conversions between systems

- Conversion from decimal to binary through division by powers of 2:
  - \(374 \div 256 = 1\)
  - \(118 \div 64 = 1\)
  - \(54 \div 32 = 1\)
  - \(22 \div 16 = 1\)
  - \(6 \div 4 = 1\)
  - \(2 \div 1 = 1\)
  - \(0 = 0\)
  - RESULT 101110110

- As an alternative, one can take the rest of successive divisions by 2:
  - \(374 \div 2 = 187\)
  - \(93 \div 46 = 2\)
  - \(23 \div 11 = 2\)
  - \(6 \div 3 = 2\)
  - \(1 = 1\)

Coding characters

- ASCII character set (American standard code for information interchange) represents characters with 7 bits. Often also written as 2 or 3 octal digits.
- In C, it corresponds to the types `char` or `unsigned char`.
- In Assembly can be coded directly with `秩序` or with octal `mov 0140, %r1`
- Anything beyond this has coding and standardization problems.

©Gustavo Alonso, ETH Zürich.
Boolean operations

- With 2 bits:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>and</th>
<th>or</th>
<th>nand</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- All possible operations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Logical</th>
<th>Space</th>
<th>Operation</th>
<th>Instruction</th>
<th>%0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>b</td>
<td>cl</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>and</td>
<td>ad</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>not</td>
<td>nd</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>xor</td>
<td>ox</td>
<td>0</td>
</tr>
</tbody>
</table>

- NOT can be simulated

- Synthetic operations such as mov are also implementable with boolean operations

<table>
<thead>
<tr>
<th>%g0, source_reg, dest_reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- or (clear):

- or %g0, %g0, dest_reg

Test operation

- Imagine you have to write in assembly something like:

```assembly
cmp %l1, 0
ble next
add %l2, 1, %l2
```

- A first approach could be (assume a is in %l1, and b in %l2):

```assembly
subcc %l1, %g0, %g0
ble next
add %l2, 1, %l2
```

- Translating the cmp (which is a synthetic operation) we get:

```assembly
orcc %l1, %g0, %g0
```

- tst is synthetic and is translated as

```assembly
orcc %l1, %g0, %g0
```

- All these testing instructions affect the condition code bits

Setting, clearing and testing flags

- Bits are often used to represent boolean flags. To manipulate these flags, the assembly language provides a number of synthetic operations

  - `bset = or (bit set)`
  - `bcclr = andn (bit clear)`
  - `btog = xor (bit toggle)`

- To check whether one or more flags are set, `btst` is used. For instance, to check whether flags 0x10 and 0x8 are set, we use:

```assembly
btst 0x18, %l1
```

  - which is translated into

```assembly
andcc %l1, 0x18, %g0
```

Adding numbers in binary

- This is one of the principles of RISC architectures: use very simple, very efficient building blocks. More complicated or fancier operations are not implemented by the processor but translated by the assembler into the simple instructions supported by the processor.

- Do not forget that the only reason why we need more complicated and fancier instructions is for the convenience of the programmer

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>sum</th>
<th>carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- When we want to add two binary numbers of a single bit each, addition is done in a similar manner:

```plaintext
3 7 7 + 6
4 1 9 +
```

- The interesting aspect is that the sum is the same as an exclusive or (XOR) and the carry is the same as an AND. Hence, we can implement addition of binary numbers as a sequence of XOR and AND operations

```plaintext
3 7 7 + 9
4 1 9 +
```
Addition as a sequence of AND and XOR

- The following C program illustrates how this works:

```c
add (int a, int b){
    int s, c; /* s is the sum, c is the carry */
    s = a^b; /* sum is xor */
    carry 001001
    while (c = (a&b) << 1){
        a = s; b = c; s = a^b;
        carry 000100
        c = (a&b) << 1; /* c = a AND b, shift the bits of c one position to the left */
    }
    return(s);
}
```

Addition in hardware

- Addition in hardware diagrams:
  - Figure 4.1: Half Adder
  - Figure 4.2: Full Adder

Modulus arithmetic

- In computers, the size of a number is limited by the size of the storage space dedicated to it.
- Because of the storage constraints, computers often work with modulus arithmetic.
- Modulus arithmetic considers only numbers in a range \([0, M]\), that is, a number \(n\) must be such that \(0 \leq n < M\). \(M\) is the modulus.
- In modulus arithmetic, once we reach the largest possible number \((M-1)\) we start again from the beginning \((0)\).
- When we perform an operation on two numbers and the results exceeds the modulus, we say that an overflow has occurred. For instance, if we work modulus 16 in binary arithmetic and we add 11 + 7:

\[
\begin{align*}
11 & = 1011 \\
7 & = 0111 + \\
\text{result} & = 0010 \\
\end{align*}
\]

The result can only be interpreted in terms of Modulus arithmetic:

\[11 + 7 = 18 = 16 + 2\]

Complements

- Complement is important because it can be used to turn subtraction into a simple sum:
  \[a - b = a + (\bar{b} + 1)\]
  where \(\bar{b} + 1\) is the diminished radix complement,
  and \(\bar{b} + 1\) is the radix complement for numbers with \(n\) digits using a system with base \(r\).
- For example, in decimal \((r=10)\) and with 2 digits \((n=2)\):
  \[23 - 07 = 23 + (99 - 07) + 1 = 23 + 92 + 1 = 23 + 93 = 16\]

- We will use complements to implement subtraction.
- and 93 is the radix complement of 07.
Subtraction by complementing

- Let b be a binary number of, e.g., 4 bits:
  - $2^4 - 1 - b$ is the one's complement of number b (same as subtracting from all 1s)
  - $2^4 - 1 - b + 1$ is the two's complement of number b (is the one's complement plus one)

- To perform $a - b$, we add a and the two's complement of b:

  
  $$
  \begin{align*}
  4 - 2 \\
  4 &= 0100 \\
  2 &= 0010 \\
  \text{two's complement}(2) &= 1110 \\
  1' &s = 0001 \\
  2' &s = 0010 = -2 \\
  \end{align*}
  $$

- This also works for negative numbers:

  
  $$
  \begin{align*}
  2 - 4 \\
  2 &= 0010 \\
  4 &= 0100 \\
  \text{two's complement}(4) &= 1110 \\
  0010 + \hline \\
  1110 &
  \end{align*}
  $$

- Try this at home!!

The devil is in the details ...

- Modulus arithmetic and complements allow us to simplify arithmetic operations and cope with the limited space ...

- ... but we have to know how to interpret the result:
  - N (negative) bit = set if most significant bit is 1
  - Z (zero) bit = set if all bits are 0
  - V (overflow) bit, set if:
    - in $c = a - b$, a and b have different signs and c and b have the same sign
    - in $c = a + b$, a and b have the same sign but different from c

- Conditions top use with signed arithmetic:

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Signed Arithmetic</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>blt</td>
<td>Branch on less</td>
<td>(N xor V) = 1</td>
</tr>
<tr>
<td>ble</td>
<td>Branch on less or equal</td>
<td>Z or (N xor V) = 1</td>
</tr>
<tr>
<td>be</td>
<td>Branch on equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>bne</td>
<td>Branch on not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>bgeu</td>
<td>Branch on greater or equal</td>
<td>(N xor V) = 0</td>
</tr>
<tr>
<td>bgu</td>
<td>Branch on greater</td>
<td>Z or (N xor V) = 0</td>
</tr>
</tbody>
</table>

- compare two numbers is implemented as subtraction, e.g., compare -8 and 6.

  $$
  \begin{align*}
  -8 - 6 \\
  -8 &= 1000 \\
  6 &= 0110 \\
  \text{two's complement}(6) &= 1010 \\
  1000 + \hline \\
  1010 &= 0010 \\
  \end{align*}
  $$

  - Overflow (when overflow occurs, the resulting sign is the complement of the real sign)

Unsigned arithmetic

- Unsigned arithmetic is, from the hardware point of view, identical to signed arithmetic

- Overflow is now indicated by the C (carry) bit, which is set when there is a carry out of the most significant bit

  $$
  \begin{align*}
  8 + 8 \\
  8 &= 1000 \\
  8 &= 1000 + \hline \\
  16 &= 1 \\
  \text{Carry set (addition)}
  \end{align*}
  $$

- Subtraction is also performed using two's complement, the C bit is set if there is no carry out of the most significant bit ($C = 0$ indicates the result is negative)

  $$
  \begin{align*}
  12 - 3 \\
  12 &= 1100 \\
  3 &= 0011 \\
  \text{two's complement}(3) &= 1111 \\
  1100 + \hline \\
  1001 &
  \end{align*}
  $$

  - There is a carry, then C bit is not set (result is positive)

  $$
  \begin{align*}
  3 - 12 \\
  3 &= 0011 \\
  12 &= 1100 \\
  \text{two's complement}(12) &= 0100 \\
  0011 + \hline \\
  0111 &
  \end{align*}
  $$

  - There is no carry, then the C bit is set (result negative)

Branching with unsigned arithmetic

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Unsigned Arithmetic</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bltu</td>
<td>Branch on less</td>
<td>C = 1</td>
</tr>
<tr>
<td>bleu</td>
<td>Branch on less or equal</td>
<td>C = 1 or Z = 1</td>
</tr>
<tr>
<td>be</td>
<td>Branch on equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>bne</td>
<td>Branch on not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>bgeu</td>
<td>Branch on greater or equal</td>
<td>C = 0</td>
</tr>
<tr>
<td>bgu</td>
<td>Branch on greater</td>
<td>C = 0 and Z = 0</td>
</tr>
</tbody>
</table>
Multiplication (decimal)

```
23
x  32
46
69
736
```

- **Multiplicand**: 23
- **Multiplier**: 32
- **Last digit of multiplier x multiplicand**: 2 x 23 = 46
- **Second to last digit of multiplier x multiplicand, shifted to the left one position**: 3 x 23 = 69 shift left(1) = 69 x 10 = 690
- **Add them together**:

```
736
```

---

Multiplication (decimal) with registers

```
00
46
```

- **Multiply**: 00 x 46 = 0000
- **Add**: 0000 + 0011
- **Shift right 1**: 0001
- **Multiply**: 0001 x 0101 = 0000
- **Add**: 0000 + 0011
- **Shift right 1**: 0000

---

Multiplication (binary) [1]

```
0011
```

- **Multiply**: 0011 x 0001 = 0000
- **Add**: 0000 + 0011
- **Shift right 1**: 0000

---

Multiplication (binary) [2]

```
0011
```

- **Multiply**: 0011 x 0000 = 0000
- **Add**: 0000 + 0011
- **Shift right 1**: 0000
Multiplication (binary) [3]

3x5
3 = 0011
5 = 0101

Product    Multiplier    Multiplicand
0000
0011
multiply

add

shift right 1

Product    Multiplier
0001
1110

Multiplication (binary) [4]

3x5
3 = 0011
5 = 0101

Product    Multiplier    Multiplicand
0001
0000
multiply

add

shift right 1

Product    Multiplier
0000
1111

Multiplication (decimal, signed)

23 x -32
10’s complement(32) = 100 - 1 - 32 + 1 = 68
23 x 68 = 1564

but the result should be -736 or, in complement form, 9264. What happened?

23 x -32 = 23 x (100 - 32)
= 23 x 100 - 23 x 32

the result is too big by 23 x 100
The problem is that the complement should be calculated based on the size of the result, not of the multiplier. Using the register multiplication algorithm, this only happens when the multiplier is negative.

23 x -32
10’s complement(32) = 10000 - 1 - 32 + 1 = 9968
23 x 9968 = 229264 modulus 10000
= 9264

which is what we wanted. An alternative to get to the same result is to subtract the excess (23 x 100)

1564 - (23 x 100)
10’s complement(2300) = 7700
1564 + 7700 = 9264

again the result we wanted. This second solution has the advantage of requiring registers of smaller size.

Multiplication signed (binary) [I]

3x-5
3 = 0011
-5 = 1011

Product    Multiplier    Multiplicand
0000
0011
multiply

add

shift right 1

Product    Multiplier
0001
1101
**Multiplication signed (binary ) [2]**

3x5
3 = 0011
-5 = 1011

**Multiplication (binary ) [3]**

3x5
3 = 0011
-5 = 1011

**Multiplication (binary ) [4]**

3x5
3 = 0011
-5 = 1011

**Division (decimal)**
DIVISION IMPLEMENTED AS SUBTRACTION

737 / 32
D = 737
1. partial = 0; d = 32 \times 10^y /* y such that d equals D in digits */
2. X = D  
3. while ((X - d) > 0)
   result += 10^y  
   partial += d  
4. result -= 10^y; partial -= d /* The loop executed one too many times */
5. D = D - partial  
6. if (D > 32) go to 1

at the end, result contains the division and D the remainder
Division (binary) with registers

\[
\begin{align*}
737 / 32 & = 01011100001 \\
32 & = 0100000 \\
\text{two's complement} (32) & = 1100000 \\
\end{align*}
\]

RESULT 1011

Non-restoring division

- Every time we go too far subtracting, we need to add the divisor back, shift it and subtract again. This can be simplified as follows.
- Let \( a \) be the dividend and \( b \) the divisor. When the result of subtracting is negative, the next step is:
  \[
  a - b + b - b/2
  \]
- working on that expression we get
  \[
  a - b + (2b - b)/2
  \]
  \[
  a - b + b/2
  \]
- that is, when we get a negative result, we do not need to add the divisor back. It is enough to shift the divisor and add it instead of subtracting.