Arithmetic / Logic Unit – ALU Design

Presentation F

Slides by Gojko Babić

07/19/2005

Functioning of 32-bit ALU

<table>
<thead>
<tr>
<th>ALU Control lines</th>
<th>Function</th>
<th>AInvvert</th>
<th>BInvert</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>subtract</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>slt</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>nor</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

- Result lines provide result of the chosen function applied to values of A and B
- Since this ALU operates on 32-bit operands, it is called 32-bit ALU
- Zero output indicates if all Result lines have value 0
- Overflow indicates a sign integer overflow of add and subtract functions; for unsigned integers, this overflow indicator does not provide any useful information
- Carry out indicates carry out and unsigned integer overflow

Designing 32-bit ALU: Beginning

1. Let us start with and function
2. Let us now add or function

Operation = 0 \rightarrow \text{and} = 1 \rightarrow \text{or}
**Designing 32-bit ALU: Principles**

- Number of functions are performed internally, but only one result is chosen for the output of ALU.
- 32-bit ALU is built out of 32 identical 1-bit ALU's.

**32-bit Adder**

This is a ripple carry adder.

The key to speeding up addition is determining carry out in the higher order bits sooner. Result: Carry look-ahead adder.

**Designing Adder**

- 32-bit adder is built out of 32 1-bit adders.

**1-bit Adder Truth Table**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

From the truth table and after minimization, we can have this design for CarryOut.
**32-bit Subtractor**

\[ A - B = A + (-B) = A + \overline{B} + 1 \]

**32-bit Adder/Subtractor**

Binvert = 0 → addition  
= 1 → subtraction

**2’s Complement Overflow**

2’s complement overflow happens:
- if sum of two positive numbers results in a negative number
- if sum of two negative numbers results in a positive number

Other 1-bit ALUs, i.e. non-most significant bit ALUs, are not affected.
### 32-bit ALU With 4 Functions and Overflow

**Control lines**

<table>
<thead>
<tr>
<th>Function</th>
<th>Binvert (1 line)</th>
<th>Operation (2 lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>subtract</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

- **Set Less Than (slt) Function**
  - slt function is defined as:
    
    \[
    A \text{ slt } B = \begin{cases} 
    000 \ldots 001 & \text{if } A < B, \ i.e. \ A - B < 0 \\ 
    000 \ldots 000 & \text{if } A \geq B, \ i.e. \ A - B \geq 0 
    \end{cases} 
    \]
  - Thus each 1-bit ALU should have an additional input (called “Less”), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
  - For the least significant bit Less value should be sign of A – B

- Missing: slt & nor functions and Zero output

### 32-bit ALU With 5 Functions

- 1-bit ALU for non-most significant bits
- 1-bit ALU for the most significant bits

### 32-bit ALU with 5 Functions and Zero

**Control lines**

<table>
<thead>
<tr>
<th>Function</th>
<th>Binvert (1 line)</th>
<th>Operation (2 lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>subtract</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>slt</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

- Operation = 3 and Binvert = 1 for slt function
- Add correction for CarryOut
32-bit ALU with 6 Functions

A \text{nor} B = \overline{A} \text{ and } \overline{B}

<table>
<thead>
<tr>
<th>Function</th>
<th>AInvert</th>
<th>BInvert</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>subtract</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>slt</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>nor</td>
<td>1</td>
<td>1</td>
<td>00</td>
</tr>
</tbody>
</table>

Figure B.5.10 (Top)

Figure B.5.10 (Bottom)

Add correction for CarryOut

Carry Out + Binvert

32-bit ALU Elaboration

- We have now accounted for all but one of the arithmetic and logic functions for the core MIPS instruction set. 32-bit ALU with 6 functions omits support for shift instructions.
- It would be possible to widen 1-bit ALU multiplexer to include 1-bit shift left and/or 1-bit shift right.
- Hardware designers created the circuit called a barrel shifter, which can shift from 1 to 31 bits in no more time than it takes to add two 32-bit numbers. Thus, shifting is normally done outside the ALU.
- We now consider integer multiplication (but not division).

Multiplication

- Multiplication is more complicated than addition:
  – accomplished via shifting and addition
- More time and more area required
- Let’s look at 3 versions based on elementary school algorithm
- Example of unsigned multiplication:

  5-bit multiplicand: $10001_2 = 17_{10}$
  5-bit multiplier: $\times 10011_2 = 19_{10}$

  | 10001 |
  | 1001  |
  | 0000  |
  | 0000  |
  | 10001 |

  $101000011_2 = 323_{10}$

  - But this algorithm is very impractical to implement in hardware

Multiplication: Example

- The multiplication can be done with intermediate additions.
- The same example:

  multiplicand: $10001_2$
  multiplier: $\times 10011_2$

  intermediate product: $10001_2$ $000000000_2$
  add since multiplier bit=1
  $0000000000_2$

  intermediate product: $10001_2$ $0000010000_2$
  shift multiplicand and add since multiplier bit=1
  $00000100001_2$

  intermediate product: $10001_2$ $00000110001_2$
  shift multiplicand and no addition since multiplier bit=0
  shift multiplicand and add multiplier since bit=1
  final result: $0101000011_2$
Multiplication Hardware: 1st Version

1. Test Multiplier0
   1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
Start
Multiplier0 = 0
Multiplier0 = 1
No: < 32 repetitions
Yes: 32 repetitions
Done

Multiplication Hardware: 2nd Version

1. Test Product0
   1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
Start
Product0 = 0
Product0 = 1
No: < 32 repetitions
Yes: 32 repetitions
Done

Multiplication Hardware: 3rd Version

• A simple algorithm:
  – Convert to positive integer any of operands (if needed) and remember original signs
  – Perform multiplication of unsigned numbers using the existing algorithm and hardware
  – Negate product if original signs disagree
• Fast multiplication algorithms.

Multiplication of Signed Integers

• A simple algorithm:
  – Convert to positive integer any of operands (if needed) and remember original signs
  – Perform multiplication of unsigned numbers using the existing algorithm and hardware
  – Negate product if original signs disagree
• This algorithm is not simple to implement in hardware, since it has to:
  – account in advance about signs,
  – if needed, convert from negative to positive numbers,
  – if needed, convert back to negative integer at the end
• Fast multiplication algorithms.
Real Numbers

- **Conversion from real binary to real decimal**
  \[-1101.1011_2 = -13.6875_{10}\]
  since: \[1101_2 = 2^3 + 2^2 + 2^0 = 13_{10}\] and
  \[0.1011_2 = 2^{-1} + 2^{-3} + 2^{-4} = 0.5 + 0.125 + 0.0625 = 0.6875_{10}\]

- **Conversion from real decimal to real binary:**
  \[+927.45_{10} = +1110011111.011100 \ldots\]

  
  - \[927/2 = 463 + \frac{1}{2} \text{ LSB}\]
  - \[0.45 \times 2 = 0.9\]
  - \[463/2 = 231 + \frac{1}{2}\]
  - \[0.9 \times 2 = 1.8\]
  - \[231/2 = 155 + \frac{1}{2}\]
  - \[0.8 \times 2 = 1.6\]
  - \[155/2 = 77 + \frac{1}{2}\]
  - \[0.6 \times 2 = 1.2\]
  - \[77/2 = 38 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]
  - \[38/2 = 19 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[19/2 = 9 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]
  - \[9/2 = 4 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[4/2 = 2 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[2/2 = 1 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]
  - \[1/2 = 0 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]

Floating Point Number Formats

- The term floating point number refers to representation of real binary numbers in computers.
- IEEE 754 standard defines standards for floating point representations
  - Single precision:
    - \[E = \text{Exp} + 127\]
    - \[927/2 = 463 + \frac{1}{2}\]
    - \[0.45 \times 2 = 0.9\]
    - \[463/2 = 231 + \frac{1}{2}\]
    - \[0.9 \times 2 = 1.8\]
    - \[231/2 = 155 + \frac{1}{2}\]
    - \[0.8 \times 2 = 1.6\]
    - \[155/2 = 77 + \frac{1}{2}\]
    - \[0.6 \times 2 = 1.2\]
    - \[77/2 = 38 + \frac{1}{2}\]
    - \[0.2 \times 2 = 0.4\]
    - \[38/2 = 19 + \frac{1}{2}\]
    - \[0.4 \times 2 = 0.8\]
    - \[19/2 = 9 + \frac{1}{2}\]
    - \[0.4 \times 2 = 0.8\]
    - \[9/2 = 4 + \frac{1}{2}\]
    - \[0.4 \times 2 = 0.8\]
    - \[4/2 = 2 + \frac{1}{2}\]
    - \[0.4 \times 2 = 0.8\]
    - \[2/2 = 1 + \frac{1}{2}\]
    - \[0.2 \times 2 = 0.4\]
    - \[1/2 = 0 + \frac{1}{2}\]
    - \[0.2 \times 2 = 0.4\]

- Double precision:
  - \[E = \text{Exp} + 1023\]
  - \[927/2 = 463 + \frac{1}{2}\]
  - \[0.45 \times 2 = 0.9\]
  - \[463/2 = 231 + \frac{1}{2}\]
  - \[0.9 \times 2 = 1.8\]
  - \[231/2 = 155 + \frac{1}{2}\]
  - \[0.8 \times 2 = 1.6\]
  - \[155/2 = 77 + \frac{1}{2}\]
  - \[0.6 \times 2 = 1.2\]
  - \[77/2 = 38 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]
  - \[38/2 = 19 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[19/2 = 9 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[9/2 = 4 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[4/2 = 2 + \frac{1}{2}\]
  - \[0.4 \times 2 = 0.8\]
  - \[2/2 = 1 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]
  - \[1/2 = 0 + \frac{1}{2}\]
  - \[0.2 \times 2 = 0.4\]

Converting to Floating Point

1. Normalize binary real number i.e. put it into the normalized form:
   \[(-1)^s \times 1.\text{Fraction} \times 2^\text{Exp}\]
   \[-1101.1011_2 = (-1)^1 \times 1.1011011 \times 2^3\]
   \[+110011111.011100 = (-1)^0 \times 1.11001111011100 \times 2^9\]

2. Load fields of single or double precision format with values from normalized form, but with the adjustment for E field.
   \[E = \text{Exp} + 127_{10} = \text{Exp} + 011111112 \text{ for single precision}\]
   \[E = \text{Exp} + 1023_{10} = \text{Exp} + 011111111112 \text{ for double precision}\]

   - E is called a biased exponent.

Floating Point: Example 1

- Find single and double precision of \[-13.6875_{10}\]
  Normalized form: \[(-1)^1 \times 1.1011011 \times 2^3\]
  - single precision:
    \[E = 112 + 011111112 = 100000102\]
    \[|1|10000010|1011011000000000000000000000000\]
  - double precision
    \[E = 112 + 011111111112 = 1000000000102\]
    \[|1|100000000010|101101100000000000000000|00000000000000000000000000000000|]
Floating Point: Example 2

- Find single and double precision of +927.4510

  Normalized form: (-1)0 \times 1.11001111101\overline{1100} \times 2^9

  - single precision
    \[ E = 1001_2 + 01111111_2 = 10001000_2 \]
    \[ 0|10001000|110011111011100110011001100... \]
    truncation \[ 0|10001000|11001111101110011001100110011001 \]
    rounding \[ 0|10001000|11001111101110011001100110011001 \]

  - double precision
    \[ E = 1001_2 + 011111111111_2 = 10000001000_2 \]
    \[ 0|10000001000|11001111101110011001 |1001100110011001100110011001100110011001100... \]
    truncation \[ 0|1001100110011001100110011001100110011001100110011001100110011001 \]
    rounding \[ 0|1001100110011001100110011001100110011001100110011001100110011001 \]

Rules for biased exponents in single precision apply only for real exponents in the range [-126,127], thus we can have biased exponents only in the range [1,254].

- The number 0.0 is represented as S=0, E=0 and Fraction=0. The infinite number is represented with E=255. There are some additional rules that are outside our scope.
- Find the largest (non-infinite) real binary number (by magnitude) which can be represented in a single precision.
  - Floating point overflow
- Find the smallest (non-zero) real binary number (by magnitude) which can be represented in a single precision.
  - Floating point underflow

Converting to Floating Point: Conclusion

Floating Point Addition

1. Compare the exponents of the two numbers. Check the smaller number to the right until its exponent would match the larger exponent.
2. Add the significands.
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
4. Round the significand to the appropriate number of bits.

Arithmetic Unit for Floating Point Addition
32-bit ALU with 6 Functions