Basic Arithmetic and the ALU

Forecast
- Representing numbers, 2’s Complement, unsigned
- Addition and subtraction
- Add/Sub ALU
  - full adder, ripple carry, subtraction, together
- Carry-Lookahead addition, etc.
- Logical operations
  - and, or, xor, nor, shifts - barrel shifter
- Overflow, MMX

Background
Recall:

n bits give rise to $2^n$ combinations

let us call a string of 32 bits as “b31 b30 . . . b3 b2 b1 b0”

No inherent meaning
- one interpretation f(b31 . . . b4 b3 b2 b1 b0) -> value
- another f(b31 . . . b4 b3 b2 b1 b0) -> control signals

Integer multiplication, division
floating point arithmetic later
not crucial for the project

32-bit types include
- unsigned integers
- signed integers
- single-precision floating point
- MIPS instructions (A.10)
Unsigned integers

\[ f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0 \]
Treat as normal binary number

e.g., 0 . . . 0110101
\[ = 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
\[ = 128 + 64 + 16 + 4 + 1 = 213 \]
max \( f(111 \ldots 11) = 2^{32} - 1 = 2^{147483647} \)
min \( f(000 \ldots 00) = 0 \)
range \([0, 2^{32} - 1]\) \(=\) # values \(2^{32}\)

Signed Integers

2's Complement

\[ f(b_{31} b_{30} \ldots b_1 b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0 \]
max \( f(0111 \ldots 11) = 2^{31} - 1 = 2147483647 \)
min \( f(100 \ldots 00) = -2^{31} = -2147483648 \) (asymmetric)
range \([-2^{31}, 2^{31} - 1]\) \(=\) # values \(2^{31} - 1 - 2^{31} + 1 = 2^{32}\)
E.g., -6
\[ \cdot 000 \ldots 0110 \rightarrow 111 \ldots 1001 + 1 \rightarrow 111 \ldots 1010 \]

Why 2's Complement

why not use signed magnitude

2's complement makes computer arithmetic simpler
just like humans don't work with Roman Numerals

Representation affects ease of calculation

Addition and Subtraction

4-bit unsigned example

\[ \begin{array}{cccc}
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 \\
\end{array} \]
\[ = 3 \\
= 10 \\
= 13 \]

4-bit 2's Complement - ignoring overflow

\[ \begin{array}{cccc}
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 \\
\end{array} \]
\[ = 3 \\
= -6 \\
= -3 \]
Subtraction

$A - B = A + 2's \text{ complement of } B$

E.g., $3 - 2$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Full Adder

full adder $(a, b, c_{in}) \rightarrow (c_{out}, s)$

$c_{out} = \text{two of more of } (a, b, c_{in})$

$s = \text{exactly one or three of } (a, b, c_{in})$

Ripple-carry Adder

Just concatenate the full adders

Ripple-carry Subtractor

$A - B = A + (-B) \Rightarrow \text{invert } B \text{ and set } C_{in0} \text{ to 1}$
Combined Ripple-carry Adder/Subtractor

control = 1 => subtract
XOR B with control and set \( C_{in0} \) to control

![Full Adder Diagram]

Carry Lookahead

The above ALU is too slow -
gate delays for add = 32 x FA + XOR \( \sim 64 \) - too slow

Theoretically:

- In parallel
  - \( \text{sum}_0 = f(c_{in}, a_0, b_0) \)
  - \( \text{sum}_i = f(c_{in}, a_i \ldots a_0, b_i \ldots b_0) \)
  - \( \text{sum}_{31} = f(c_{in}, a_{31} \ldots a_0, b_{31} \ldots b_0) \)

Carry Lookahead

Need compromise

- build tree so delay is \( O(\log_2 n) \) for \( n \) bits
  - E.g., 2 x 5 gate delays for 32-bits

We will give the basic idea with (a) 4-bit then (b) 16-bit adder

A little convoluted!
Carry Lookahead

Therefore
\[ c_1 = g_0 + p_0 \cdot c_0 \]
\[ c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \]
\[ = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \]
\[ c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]
\[ c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]

Uses one level to form \( p_i \) and \( g_i \), two levels for carry

But, this needs \( n + 1 \) fanin at the OR and the rightmost AND

---

Hierarchical Carry Lookahead for 16 bits

Build 16-bit adder from four 4-bit adders

Figure out Generate and Propagate for 4-bits together

\[ G_{0,3} = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \]
\[ P_{0,3} = p_3 \cdot p_2 \cdot p_1 \cdot p_0 \] (Notation a little different from the book)
\[ G_{4,7} = g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 \]
\[ P_{4,7} = p_7 \cdot p_6 \cdot p_5 \cdot p_4 \]
\[ G_{12,15} = g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12} \]
\[ P_{12,15} = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12} \]
Carry Lookahead Basics

Fill in the holes in G's and P's:
\[ G_{i,k} = G_{j+1,k} + P_{j+1,k} \cdot G_{i,j} \quad \text{(assume } i < j +1 < k \text{)} \]
\[ P_{i,k} = P_{i,j} \cdot P_{j+1,k} \]
\[ G_{0,7} = G_{4,7} + P_{4,7} \cdot G_{0,3} \quad P_{0,7} = P_{0,3} \cdot P_{4,7} \]
\[ G_{8,15} = G_{12,15} + P_{12,15} \cdot G_{8,11} \quad P_{8,15} = P_{8,11} \cdot P_{12,15} \]
\[ G_{0,15} = G_{8,15} + P_{8,15} \cdot G_{0,7} \quad P_{0,15} = P_{0,7} \cdot P_{8,15} \]

Carry Lookahead: Compute G's and P's

\[ G_{12,15} \quad G_{8,11} \quad G_{4,7} \quad G_{0,3} \]
\[ P_{12,15} \quad P_{8,11} \quad P_{4,7} \quad P_{0,3} \]
\[ G_{8,15} \quad P_{8,15} \]
\[ G_{0,15} \quad P_{0,15} \]

Carry Lookahead: Compute c's

\[ \begin{align*}
G_{12} - G_{15} & \quad G_{8} - G_{11} & \quad G_{4} - G_{7} & \quad G_{0} - G_{3} \\
C_{12} & \quad C_{8} & \quad C_{4} & \quad C_{0} \\
P_{12} - P_{15} & \quad P_{8} - P_{11} & \quad P_{4} - P_{7} & \quad P_{0} - P_{3} \\
C_{12} & \quad C_{8} & \quad C_{4} & \quad C_{0} \\
G_{8,11} & \quad P_{8,11} & \quad G_{0,3} & \quad P_{0,3} \\
G_{0,7} & \quad P_{0,7} & \quad G_{0,7} & \quad P_{0,7} \\
C_{0} & \quad C_{0} & \quad C_{0} & \quad C_{0} \\
\end{align*} \]

Other Adders: Carry Select

Two adds in parallel - one with C_{in} 0 and the other C_{in} 1

- When C_{in} is done, select the right result

\[ \begin{align*}
\text{Full Adder} & \quad 0 \\
\text{Full Adder} & \quad 1 \\
\text{2-1 Mux} & \quad \text{next select} \\
\text{select} & \quad c_0 \\
\end{align*} \]
Other Adders: Carry Save

A + B -> S
Save carries A + B -> S, C_{out}
Use C_{in} A + B + C -> S1, S2 (3# to 2# in parallel)
Used in combinational multipliers by building a Wallace Tree

Wallace Tree

Logical Operations

Bitwise AND, OR, XOR, NOR
- Implement with 32 gates in parallel
Shifts and rotates
- rol -> rotate left (MSB --> LSB)
- ror -> rotate right (LSB --> MSB)
- sll -> shift left logical (0 --> LSB)
- srl -> shift right logical (0 --> MSB)
- srl -> shift right arithmetic (old MSB --> new MSB)

Shifter

E.g., Shift left logical for d<7:0> and shamt<2:0>
Using 2-1 Muxes called Mux(select, in_0, in_1)
stage0<7:0> = Mux(shamt<0>, d<7:0>, 0 || d<7:1>)
stage1<7:0> = Mux(shamt<1>, stage0<7:0>, 00 || stage0<6:2>)
dout<7:0> = Mux(shamt<2>, stage1<7:0>, 0000 || stage1<3:0>)
For Barrel shifter used wider muxes
Shifter

<table>
<thead>
<tr>
<th>Mux</th>
<th>stage0</th>
<th>shift based on 0th bit by 0 or 1</th>
<th>shamt0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s0_7</td>
<td>s0_0</td>
<td></td>
</tr>
<tr>
<td>Mux</td>
<td>stage1</td>
<td>shift based on 1st bit by 0 or 2</td>
<td>shamt1</td>
</tr>
<tr>
<td></td>
<td>s1_7</td>
<td>s1_0</td>
<td></td>
</tr>
<tr>
<td>Mux</td>
<td>dout</td>
<td>shift based on 2nd bit by 0 or 4</td>
<td>shamt2</td>
</tr>
<tr>
<td></td>
<td>dout_7</td>
<td>dout_7</td>
<td></td>
</tr>
</tbody>
</table>

All Together

Overflow

with n-bits only $2^n$ combinations

Unsigned [0, $2^n - 1$], 2's Complement [-$2^{n-1}$, $2^{n-1} - 1$]

Unsigned Add

$5 + 6 > 7$

101

+ 110

1011

$f(3:0) = a(2:0) + b(2:0) \Rightarrow \text{overflow} = f(3) ;; \text{carryout}$

Overflow

More involved for 2's Complement

-1 + -1 = -2

111

+ 111

1110

110 = -2 is correct => can't just use carry-out
Addition Overflow

When is overflow NOT possible? p1, p2 > 0 and n1, n2 < 0

p1 + p2
not possible

p1 + n1
not possible

n1 + p2
not possible

n1 + n2
overflow = X * a(2) * b(2) + Y * a(2) * b(2)

What are X and Y?

2 + 3 = 5 > 4
010

+ 011
101 = -3 < 0!
In general, X = f(2)

-1 + -4
111

+ 100
011 which is 011 > 0
In general Y = f(2)

Overflow = f(2) * a(2) * b(2) + f(2) * a(2) * b(2)

Subtraction Overflow

No overflow on a-b if signs are same

neg - pos ==> neg ;; overflow otherwise

pos - neg ==> pos ;; overflow otherwise

overflow = f(2) * a(2) * b(2) + f(2) * a(2) * b(2)

What to do on overflow

Ignore!

Flag - condition code that may be tested by software

sticky flag - e.g., for floating point

trap - possibly with mask
Zero and Negative

zero = f(2) + f(1) + f(0)
can't also look at f(3) because

001 +1
+ 111 -1
1000 0
So, negative = f(2)

Zero and Negative

May also want correct answer even on overflow
negative = (a < b) = (a-b < 0) even if overflow
E.g., is -4 < 2?

100 -4
-010 2
1010 -6 => overflow
If you work it out,
negative = f(2) XOR overflow

MMX

MMX [Peleg & Weiser, IEEE Micro, Aug. 96]
- Goal 2x performance in audio, video, etc.
- Key technique: SIMD - single instruction multiple data
- 1999 Streaming SIMD Extensions in same spirit

Data types
1 x 64 bit quad word
2 x 32 bit double-word
4 x 16 bit word
8 x 8 bit byte

MMX, cont.

E.g., ADDB (for byte)

17 87 100 ... 6 more
+ 17 13 200 ... 6 more
34 100 44 == 300 mod 256
or 255 == maximum value

E.g., 16 element dot product from matrix multiply
- [a1... a16] x [b1 ... b16] = a1*b1 + ... + a16*b16
- IA-32: 32 loads, 16 *, 15 +, 12 loop control = 76 instr.
- MMX: 16 instr.
- Cycles 200 for int, 76 for FP, & 12 for MMX (6x over FP!)
MMX, cont.

Others: MOV, (UN)PACK, & MASK (e.g., next)

<table>
<thead>
<tr>
<th>15</th>
<th>15</th>
<th>100</th>
<th>120</th>
<th>101</th>
<th>76</th>
<th>15</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>FF</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

Why? Weatherperson at 00’s & weathermap at FF’s

Comments

- Backward compatible & no OS changes (overload FP regs)
- Others have similar: Sun, HP, and now Intel SSE
- ISVs (i.e., for games) have not (yet) embraced