SWITCHING THEORY

Twan Basten
Ralph H.J. M. Otten
Eindhoven University of Technology

course 5A050
september-november 2004

overview

- you have read in “introduction to logic design”:
  - chapter 1.1: number systems
  - chapter 5: arithmetic circuits, multiplexers, gate arrays

- part five:
  - positional number notations
  - negative-number representations
  - addition, subtraction
  - overflow conditions

- part six:
  - arithmetic circuits
  - arithmetic logic units

motivation

- time vs. space trade-offs
  - doing things fast requires more logic and thus more space
  - example: carry lookahead logic

- arithmetic logic units
  - critical component of processor datapath
  - core of most computer instructions

we need to understand number systems

positional number notation

- base 10 (decimal)
  \[154_{10} = 1 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 = 1 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 = 154\]

- base 2 (binary)
  \[10011010 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 128 + 16 + 8 + 2 = 154\]

- base 8 (octal; 0, 1, 2, 3, 4, 5, 6, 7)
  \[232_8 = 2 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 2 \times 64 + 3 \times 8 + 2 \times 1 = 128 + 24 + 2 = 154\]

- base 16 (hexadecimal; 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
  \[9A_{16} = 9 \times 16^1 + 10 \times 16^0 = 144 + 10 = 154\]

- base 3 (ternary; 0, 1, 2)
  \[1220_3 = 1 \times 3^3 + 2 \times 3^2 + 2 \times 3^1 + 0 \times 3^0 = 1 \times 27 + 2 \times 9 + 2 \times 3 + 0 \times 1 = 81 + 54 + 18 + 0 = 154\]
binary 2 octal 2 hexadecimal

remember: $154_{10} = 10011010_2 = 232_8 = 9A_{16}$

bit grouping

\[
\begin{align*}
10011010 \quad &\quad 2\quad 3\quad 2 \\
2\quad 3\quad 9\quad A &\quad 16
\end{align*}
\]

why does this work?

Assume $n = 4k$ bits binary number $\rightarrow k$ digits hex number

\[
\begin{align*}
&\quad a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + a_{n-3}2^{n-3} + a_{n-4}2^{n-4} = 2^{4k-1} \\
&\quad + \ldots + a_{3}2^{3} + a_{2}2^{2} + a_{1}2^{1} + a_{0}2^{0} \\
&\quad = (a_{n-1}2^{3} + a_{n-2}2^{2} + a_{n-3}2^{1} + a_{n-4}2^{0})2^{4(k-1)} \\
&\quad + \ldots + (a_{3}2^{3} + a_{2}2^{2} + a_{1}2^{1} + a_{0}2^{0})2^{4(0)} \\
&\quad = b_{k-1}16^{k-1} + \ldots + b_{0}16^{0} \quad \text{with } b_{i} \text{ created with bit grouping}
\end{align*}
\]

reverse transformation works as well

base 10 to base X: successive division

\[
\begin{align*}
154_{10} \quad &\quad 154 / 3 = 51 \text{ remainder } 1 \\
&\quad 51 / 3 = 17 \text{ remainder } 0 \\
&\quad 17 / 3 = 5 \text{ remainder } 2 \\
&\quad 5 / 3 = 1 \text{ remainder } 2 \\
&\quad 1 / 3 = 0 \text{ remainder } 1
\end{align*}
\]

why does this work?

Assume $N = (a_{n-1}a_{n-2} \ldots a_{2}a_{1}a_{0})_{10}$

\[
\begin{align*}
Q_0 &= a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_{1}2^{1} + a_{0}2^{0} \quad \text{remainder } a_{0} \\
Q_1 &= a_{n-1}2^{n-3} + a_{n-2}2^{n-4} + \ldots + a_{2}2^{0} \quad \text{remainder } a_{1} \\
&\quad \vdots \\
Q_{n-1} &= a_{n-2}2^{0} \quad \text{remainder } a_{n-1}
\end{align*}
\]

overview conversions
addition / subtraction

addition base 3

\[
\begin{array}{c}
1 & 1 & 1 \\
1 & 2 & 2 & 0 & 1 \\
\hline
2 & 1 & 2 \\
\hline
2 & 0 & 1 & 2 & 0
\end{array}
\]

carry

\[
(154 + 23 = 177)
\]

\[
(=6 + 9 + 162 = 177)
\]

subtraction base 3

\[
\begin{array}{c}
1 & 2 & 2 & 0 & 1 \\
-2 & -1 & -2 \\
\hline
1 & 1 & 2 & 1 & 2
\end{array}
\]

borrow

\[
(154 - 23 = 131)
\]

\[
(=2 + 3 + 18 + 27 + 81 = 131)
\]

representation of negative numbers

- representation of positive numbers same in most systems
- major differences are in how negative numbers are represented
- three major schemes:
  - sign and magnitude
  - ones complement
  - twos complement
- assumptions:
  - 4 bit machine words
  - 16 different values can be represented
  - roughly half are positive, half are negative

sign and magnitude representation

- high order bit is sign: 0 = positive (or zero), 1 = negative
- three low order bits is the magnitude: 0 (000) thru 7 (111)
- number range for n bits = +/- (2^n-1)
- 2 representations for 0
- cumbersome addition/subtraction

sign and magnitude: addition and subtraction

- general remark
  - subtraction can be implemented via addition and negation:
  - \( 4 - 3 = 4 + (-3) \)
- addition of two numbers with equal signs:
  - straightforward
  - \[
  \begin{align*}
  \text{Addition of magnitude:} & \quad 1 + 3 = 0001 + 0011 = 1011 \\
  \text{Result sign equals operands' sign:} & \quad 1 + 3 = 1000 \text{ (positive)}
  \end{align*}
  \]
- addition of two numbers with different signs:
  - how do we implement this?
  - \[
  \begin{align*}
  \text{Subtract smallest from greatest magnitude:} & \quad 4 - 3 = 0100 - 0011 = 1100 \\
  \text{Take sign of greatest magnitude:} & \quad 4 - 3 = 1001
  \end{align*}
  \]
  - need subtractor and comparator
ones complement

high order bit is sign: 0 = positive (or zero), 1 = negative
number range for n bits = +/-2^(n-1)
positive numbers: low order bits are the magnitude
negative numbers: low order bits are the offset from -2^(n-1)
2 representations for 0
complications with addition

ones complement: addition and subtraction

as before, subtraction is implemented via addition and negation
addition of two numbers with equal signs:

straightforward addition
\[
\begin{align*}
  1 & \rightarrow 0001 \\
  -1 & \rightarrow 1110 \\
  3 + & \rightarrow 0011 \\
  4 + & \rightarrow 0100 \\
  3 + & \rightarrow 1100 \\
  -4 + & \rightarrow 1110 \\
  1010 & \rightarrow +(-5)
\end{align*}
\]
end-around carry

addition of two numbers with different signs:

\[
\begin{align*}
  4 & \rightarrow 0100 \\
  -4 & \rightarrow 1011 \\
  -3 + & \rightarrow 1100 \\
  3 + & \rightarrow 0011 \\
  1 + & \rightarrow 0110 \\
  1011 & \rightarrow +1
\end{align*}
\]
adder and bit-wise complement suffice to implement addition and subtraction

ones complement

let m be a positive number
what is the 1's complement representation of -m?
let \( \bar{m} \) be this representation

\[
\begin{align*}
  \bar{m} &= (2^n - 1 - m)_{1c} \\
  \text{works also for negative numbers!}
\end{align*}
\]
example: 1’s complement of -7

\[
\begin{align*}
  (7)_{10} &= 10000_2 \\
  (1)_{10} &= 00001_2 \\
  \text{(7)}_{10} &= 0111_{1c} \\
  \text{(-7)}_{10} &= 1111_{1c}
\end{align*}
\]
shortcut method:
bit-wise complement

\[
\begin{align*}
  0111 & \rightarrow 1000 \quad \text{and so does shortcut method}
\end{align*}
\]

end-around carry

why does the end-around carry work?
it is equivalent to subtracting 2^n and adding 1
\[
\begin{align*}
  (m + (k))_{10} &= ? \\
  \text{k, m positive with } m > k \\
  (k, \text{m positive}) \\
  m_{1c} + k &= m_{1c} + 2^n \cdot k_{1c} = m_{1c} + (2^n \cdot 1_2) \cdot k_{1c} \\
  m_{1c} + k &= (m + k + (2^n \cdot 1))_{10} \\
  \text{so, simply adding 1's complement representations of m and -k means that we have to subtract } 2^n \text{ and add 1 in order to be correct: end-around carry}
\end{align*}
\]

\[
\begin{align*}
  (-m + (k))_{10} &= ? \\
  \text{k, m positive with } m + k < 2^{n-1} \\
  m_{1c} + k &= (2^n \cdot 1_2 \cdot m_{1c} + (2^n \cdot 1_2 \cdot k_{1c}) = (2^n \cdot 1_2 + [2^n \cdot 1_2 \cdot (m + k)_{1c}] \\
  \text{after end-around carry:}
  (m + k)_{10} &= (m + k)_{1c} + (m + k)_{10} \text{ in 1's comp!}
\end{align*}
\]
Twos complement

- High order bit is sign: 0 = positive (or zero), 1 = negative
- Number range for n bits: from $2^{n-1}$ to $-2^{n-1}$
- Positive numbers: low order bits are the magnitude
- Negative numbers: low order bits are the offset from $-2^{n-1}$

1. Twos complement, shifted 1 position clockwise
2. 1's complement, shifted 1 position clockwise
3. Representation for 0 and no complications with addition

Twos complement: addition and subtraction

As before, subtraction is implemented via addition and negation

1. Straightforward addition
   - $1 + 0001 = 1111$
   - $3 + 0011 + 1000 = 1101$
   - $4 + 0100 + 1101 + 1111$

2. Ignore carry-out
   - $4 + 0100 = 1100$
   - $-3 + 1101 + 1111$

Adder and bit-wise complement suffice to implement addition and subtraction

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

Shortcut method:
- Bit-wise complement + 1
- 0111 -> 1000 + 1 -> 1001
- (7) -> (-7)

Example: Two's complement of -7

Let $m$ be a positive number

Let $m^*$ be this representation

$m^* = (2^n)_{10} - (m)_{2c}$

Works also for negative numbers!

Example: Two's complement of -7

$(2^4)_{10} = 10002$

$(7)_{10} = 0111_{2c}$

$(7)_{10} = 1001_{2c} = (-7)_{10}$

Shortcut method:
- Bit-wise complement + 1
- 0111 -> 1000 + 1 -> 1001
- (7) -> (-7)

And so does shortcut method

Carry-out?

Why can the carry-out be ignored?

It is equivalent to subtracting $2^n$

1. $(m + (k))_{10} = ?$
   - $k, m$ positive with $m > k$
   - $(k, m$ positive $)_{10} = m_2 + (2^n)_{2c}$
   - $m_2 + k^* = m_2 + (2^n)_{2c} - k_2 = m_2 + (2^n)_{2c} - k_2 = (m - k + 2^n)_{10}$
   - So, simply adding 2's complement representations of $m$ and $-k$ means that we have to subtract $2^n$ in order to be correct: ignore carry-out

2. $((-m) + (k))_{10} = ?$
   - $k, m$ positive with $m + k \leq 2^n$
   - $m^* + k^* = (2^n)_{2c} + (m + k)_{2c} = (2^n)_{2c} + (m + k)_{2c}$
   - Ignoring carry-out:
     - $(2^n)_{2c} - (m + k)_{2c} = (m + k)^*$
   - But this is the correct form for representing $-(m + k)_{10} = ((-m) + (k))_{10}$ in 2's comp!
overflow

- Adding two positive numbers yields a negative number.
- Adding two negative numbers yields a positive number.

overflows can only occur with numbers of equal signs!

detecting overflow

- Sum of sign bits carry-out always 0!
- No overflow.
- Carry-in of 1 implies overflow.
- Carry-in of 0 implies overflow.
- Carry-out always 1!

overview

- Part five:
  - Positional number notations
  - Negative-number representations
  - Addition, subtraction
  - Overflow conditions

- Part six:
  - Arithmetic circuits
  - Arithmetic logic units

serial binary addition

- Half adder: $a_i b_i \oplus sum$ carry = $a_i \cdot b_i$

multi-bit adder: (ripple-carry)
serial binary addition

full adder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>s</th>
<th>c_{i+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ s_i = c_i \oplus a_i \oplus b_i \]

\[ c_{i+1} = b_i \cdot c_i + a_i \cdot c_i + a_i \cdot b_i = c_i \cdot (a_i + b_i) + a_i \cdot b_i \]

multi-bit adder: (ripple-carry)

adder/subtractor

\[(a + b)_{10} = (a + b)_{2c}\]

\[(a - b)_{10} = (a + (-b))_{10} = (a)_{2c} + \overline{b} + 1_{2c}\]

multi-bit adder: (ripple-carry)

worst-case carry delay

critical delay: the propagation of carry from low to high order stages

assumption: all gates have delay 1

two gate delays to compute \(c_{i+1}\)

\[c_{i+1} = \text{late arriving signal} \rightarrow a \oplus b \rightarrow @n c_i \rightarrow @n+1 (\text{if } n > 0)\]

\[c_{i+1} = \text{one gate delay to compute } s_i\]

\[s_i = \text{final sum and carry}\]

worst-case carry delay

critical delay: the propagation of carry from low to high order stages

observed delay depends on bit patterns!

32 bits? 64 bits?

solution: carry lookahead logic (not treated)

trade-off between space and time

1111 + 0001 worst case addition

32 bits? 64 bits?

observed delay depends on bit patterns!
**arithmetic logic units**

unit implementing logical and arithmetic operations

inputs: \( a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_0 \)

outputs: \( f_0, \ldots, f_{n-1}, f_n = c_n \)

control inputs: \( m, s_0, s_1 \)

slice \( i \): 6 inputs \( m, s_0, s_1, a_i, b_i, c_i \) and 2 outputs \( f_i, c_{i+1} \)

\( S_1, S_2 \)

function comment

\( m = 0, s_0, s_1 = 0 \)

logical bitwise operations

\( m = 1, s_0 = 0 \)

arithmetic operations


```
.. model alu.espresso
    .inputs m s0 s1 ai bi
    .outputs fi co
    .names m ci co [30] [33] [35] fi
    110--  1
    -11-1  1
    -01-1  1
    -00-0  1
    .names m ci [30] [33] co
    -1-1  1
    -11  1
    111  1
    .names s0 ai [30]
    01 1
    10 1
    .names m s1 bi [33]
    111 1
    .names s1 bi [35]
    0- 1
    -0 1
    .end
```

12 gates

```
      a1
       ^
      /   \
   a2  / \
     |   |
     v   v
    a3  a4
         |   |
         v   v
      x1  x2
        ^   \
        |    |
        |    v
        c0  c1
```

8 gates (but 3 are xor)

```
S_1 = 0 blocks b_i, (operations involve a_i only)

\( m = 0 \) blocks \( c_i \)

arithmetic mode (\( m = 1 \)):

- \( c_i \) to xor gate \( x_2 \)
- if \( S_1 = 1 \), \( b_i \) to xor gate \( x_3 \)
- if \( S_0 = 0 \), \( x_1 \) passes \( a_i \)
- if \( S_0 = 1 \), \( x_1 \) passes \( a_i \)

inputs of \( o_1 \) are \( a_i, c_i, b_i \)

logic mode (\( m = 0 \)):

- cascaded xors form output \( f_i \) from \( a_i \) and \( b_i \)
- output \( c_{i+2} \) don't care
```