Chapter 4 – Binary Arithmetic

These are lecture notes to accompany the book

SPARC Architecture, Assembly Language Programming, and C,


By Michael Weeks
Arithmetic

- Arithmetic involves
  - addition
  - subtraction
  - multiplication
  - division

- People use base 10
- Computers use base 2
- Base 2 is actually *easier* than base 10
Addition

- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column on the left

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Carry</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Carry</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
# Binary Sums and Carries

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Sum</th>
<th>a</th>
<th>b</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<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>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**XOR**

**AND**

Richard P. Paul, *SPARC Architecture, Assembly Language Programming, and C*
Binary Addition Example

mov 0x45, %r1
mov 0xE7, %r2
add %r1, %r2, %r3

- What values are in the registers after this?

0x45 = \[0 0100 0101\]
0xE7 = \[+ 0 1110 0111\]

\[
\begin{array}{c}
0 \quad 0100 \quad 0101 \\
+ \quad 1110 \quad 0111 \\
\hline \\
1 \quad 0010 \quad 1100
\end{array}
\]

\[\Rightarrow 0x12C\]

so r1=0x45, r2=0xE7, r3=0x12C
Half Adder / Full Adder

• A half adder
  – For adding 2 bits
  – Gives “carry out” and “sum”
  – 1 AND and 1 XOR gate

• A full adder
  – For adding 2 bits plus a “carry in”
  – Gives “carry out” and “sum”
  – 2 ANDs, 2 XORs, and 1 OR

• You’ll see more of this in the Computer Architecture class
Figure 4.2 – Full Adder
Modulus Arithmetic

• “Modulus arithmetic considers only numbers in the range $0 \leq n < M$, where $M$ is the modulus.” [page 117]

• Example: a car odometer

  if a car has 99999 miles on it,
  and you drive another mile,
  the odometer will then read 00000
Modulus Arithmetic

• Computers do this
• SPARC has 32-bit registers
• Each register can store an integer number
  – between 0 and $2^n-1$
  – where $n=32$ for SPARC
• if you have a value of $2^n-1$ in a register, and you add 1 to this, the register will then hold a value of 0
Subtraction

• Let $r$ be the base, and $n$ the number of digits
  \[ a - b = a + (r^n - 1 - b) + 1 \]
  • since the result is modulus $r^n$, adding $r^n$ does not affect the result
    – Imagine if your car’s odometer read 54321 and you drove 100,000 miles, it would then read 54321 again.
  • no borrowing is needed
  • once $(r^n - 1 - b)$ is found, subtraction can be done with addition
Complement Arithmetic

- $r^{n-1} - b$ is called the
  - “nine’s complement” if $r=10$
  - “one’s complement” if $r=2$
- $r^{n-1} - b + 1$ is called the radix complement
  - “ten’s complement” if $r=10$
  - “two’s complement” if $r=2$
- Any number where the most significant digit $\geq (r/2)$ is considered negative
  - $10000000$ means $-128$ when $r=2$, $n=8$
  - $84$ means $-16$ when $r=10$, $n=2$ [see page 120]
Two’s Complement

• In binary, finding the one’s complement and the two’s complement are easy

• One’s complement:
  – Replace every 0 with a 1,
  – and replace every 1 with a 0

• Two’s complement:
  – Find the one’s complement,
  – and add 1
Two’s Complement Example

• What is \(-16 \) (decimal) in binary \((r=2)\)?
• We’ll assume \(n=8\)

\[
16 = 0001 \ 0000 \text{ in binary}
\]

\[
1110 \ 1111 \text{ one’s complement}
\]

\[
+ \ 0000 \ 0001 \text{ add 1}
\]

\[
1111 \ 0000 \text{ two’s complement}
\]
## Two’s Complement Numbers

<table>
<thead>
<tr>
<th>Four-bit</th>
<th>Eight-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>0111 1111</td>
</tr>
<tr>
<td>0110</td>
<td>0111 1110</td>
</tr>
<tr>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0000 0100</td>
</tr>
<tr>
<td>0011</td>
<td>0000 0011</td>
</tr>
<tr>
<td>0010</td>
<td>0000 0010</td>
</tr>
<tr>
<td>0001</td>
<td>0000 0001</td>
</tr>
<tr>
<td>0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1111</td>
<td>1111 1111</td>
</tr>
<tr>
<td>1110</td>
<td>1111 1110</td>
</tr>
<tr>
<td>1101</td>
<td>1111 1101</td>
</tr>
<tr>
<td>1100</td>
<td>1111 1100</td>
</tr>
<tr>
<td>1011</td>
<td>1111 1011</td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1000 0001</td>
</tr>
<tr>
<td>1000</td>
<td>1000 0000</td>
</tr>
</tbody>
</table>
Number Ranges

• A signed number has the range
  \(-2^{n-1} \) to \(2^{n-1} - 1\)

• An unsigned number has the range
  0 to \(2^n - 1\)

• What is the range of a SPARC register?
  -2,147,483,648 to 2,147,483,647 (signed)
  0 to 4,294,967,295 (unsigned)
Addition and Subtraction

• “The two’s complement system is an interpretation of numbers in registers; the hardware always performs binary addition.” [page 122]
• To subtract, find the 2’s complement of the 2nd operand, and add
• There is no need for a hardware subtractor
Multiplication and Division

• These operations can be done with additions and subtractions
• There is no instruction for multiplication (nor division) in the SPARC architecture
• We will use the “call .mul” and “call .div”
• We will skip sections 4.10 and 4.11 (but do read about numeric labels on page 137)
Shift Operations

- A registers’ contents can be shifted
  - left shift is like multiplying by 2
  - right shift is like dividing by 2
- Logical shift
  - copies 0 into most significant bit(s)
- Arithmetic shift
  - Copies the sign bit into most significant bit
    (otherwise, negatives could become positives)
Shift Instructions

- Shift right logical (srl)

- Shift right arithmetic (sra)

- Shift left logical (sll)

- Shift left arithmetic is not provided,
  - it would be the same as shift left logical
Shift Instructions

• Shift right logical (srl)
  \[\text{srl \ reg}_{rs1}, \ \text{reg}\_or\_imm, \ \text{reg}_rd\]
• Shift right arithmetic (sra)
  \[\text{sra \ reg}_{rs1}, \ \text{reg}\_or\_imm, \ \text{reg}_rd\]
• Shift left logical (sll)
  \[\text{sll \ reg}_{rs1}, \ \text{reg}\_or\_imm, \ \text{reg}_rd\]
• The number of shifts is the low 5 bits of the \text{reg}\_or\_imm, so largest shift is 31
Branching Conditions

• Branching is based on the following flags:

  N  (negative)
  the most significant bit of the result
  not used with unsigned numbers

  V  (overflow)
  when result is too big for the register

  Z  (zero)
  set when all bits of the result are 0

  C  (carry)
  set when an addition has carry out,
  or when subtraction does not have carry out
### Signed Branches

<table>
<thead>
<tr>
<th>Assembler Mnemonic</th>
<th>Signed Arithmetic</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bl</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>bge</td>
<td>branch on greater or equal</td>
<td>(N xor V) = 0</td>
</tr>
<tr>
<td>bg</td>
<td>branch on greater</td>
<td>Z or (N xor V) = 0</td>
</tr>
</tbody>
</table>
Unsigned Arithmetic

- “Hardware operations on signed and unsigned numbers are identical.” [page 124]
- The carry flag (C) can be used instead of the overflow flag (V)
# Unsigned Branches

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Signed Arithmetic</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>blu</td>
<td>branch on less</td>
<td>C = 1</td>
</tr>
<tr>
<td>bleu</td>
<td>branch on less or equal</td>
<td>Z or C = 1</td>
</tr>
<tr>
<td>be</td>
<td>branch on equal</td>
<td>Z = 1 (same as signed branch)</td>
</tr>
<tr>
<td>bne</td>
<td>branch on not equal</td>
<td>Z = 0 (same as signed branch)</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>Z = 0 and C = 0</td>
</tr>
</tbody>
</table>
## Condition Code Branches

<table>
<thead>
<tr>
<th>Assembler Mnemonic</th>
<th>Signed Arithmetic Branches</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bneg</td>
<td>branch on negative</td>
<td>N = 1</td>
</tr>
<tr>
<td>bpos</td>
<td>branch on positive</td>
<td>N = 0</td>
</tr>
<tr>
<td>bz</td>
<td>branch on zero set</td>
<td>Z = 1 (same as be)</td>
</tr>
<tr>
<td>bnz</td>
<td>branch on zero not set</td>
<td>Z = 0 (same as bne)</td>
</tr>
<tr>
<td>bvs</td>
<td>branch on overflow set</td>
<td>V = 0</td>
</tr>
<tr>
<td>bvc</td>
<td>branch on overflow clear</td>
<td>V = 1</td>
</tr>
<tr>
<td>bcs</td>
<td>branch on carry set</td>
<td>C = 1 (same as blu)</td>
</tr>
<tr>
<td>bcc</td>
<td>branch on carry clear</td>
<td>C = 0 (same as bgeu)</td>
</tr>
</tbody>
</table>
Numeric Labels

• “The assembler allows single-digit labels to appear many times in a single source file.” [page 137]

```assembly
addcc %lo_r, %lo_r, %lo_r
bcc 1f  ! find label ‘1’ forward
add %hi_r, %hi_r, %hi_r
bset 1, %hi_r
1:
```

• The letter “b” or “f” must be appended to the digit, for backward or forward direction.
• The closest label in that direction will be the branch target, if the branch is taken.
• These are good for labels used in control structures.

Richard P. Paul, *SPARC Architecture, Assembly Language Programming, and C*
Extended Precision

• What if we need to work with data that are more than 32 bits wide?

• We can use extended precision instructions, with the carry

  \[ \text{addx } \text{reg}_{rl1}, \text{reg}_or\_imm, \text{reg}_{rd} \]
  \[ \text{addxcc } \text{reg}_{rl1}, \text{reg}_or\_imm, \text{reg}_{rd} \]

these mean:

  \[ \text{reg}_{rd} = \text{reg}_{rl1} + \text{reg}_or\_imm + C \]
Extended Precision (Sub)

• We can use extended precision instructions, with the carry

\[
\text{subx} \quad \text{reg}_{rs1}, \text{reg}_{or\_imm}, \text{reg}_{rd} \\
\text{subxcc} \quad \text{reg}_{rs1}, \text{reg}_{or\_imm}, \text{reg}_{rd}
\]

these mean:

\[
\text{reg}_{rd} = \text{reg}_{rs1} - \text{reg}_{or\_imm} - C
\]

• Data larger than 32 bits must be stored in 2 (or more) registers. That is, add (or sub) the low part first, then use the extended precision addx (or subx) for the high part.

• This is just like doing arithmetic on paper, where we work with one digit at a time.
Summary of Chapter 4

- sum and carry
- modulus arithmetic
  - one’s complement
  - two’s complement
- signed and unsigned arithmetic
- condition codes
- shifting
- numeric labels
- extended precision
Summary of Chapter 4

```
bl  label

also: ble, be, bne, bge, bg,
blu, bleu, be, bne, bgeu, bgu,
bneg, bpos, bz, bnz, bvs, bvc, bcs, bcc

srl  reg\_rs1, reg\_or\_imm, reg\_rd
sra  reg\_rs1, reg\_or\_imm, reg\_rd
sll  reg\_rs1, reg\_or\_imm, reg\_rd
addx  reg\_rs1, reg\_or\_imm, reg\_rd
addxcc  reg\_rs1, reg\_or\_imm, reg\_rd
subx  reg\_rs1, reg\_or\_imm, reg\_rd
subxcc  reg\_rs1, reg\_or\_imm, reg\_rd
```