CPU Arithmetic

Overview

⇨ The Central Processing Unit (CPU) is the logical combination of the Arithmetic Logic Unit (ALU) and the system's control unit
⇨ In this section, we focus on the ALU and its operation
– Overview of the ALU itself
– Data representation (review!)
– Computer Arithmetic and its H/W implementation

The ALU

⇨ The ALU is that part of the computer that actually performs arithmetic and logical operations on data
⇨ All other elements of the computer system are there mainly to bring data to the ALU for processing or to take results from the ALU
⇨ Registers are used as sources and destinations for most ALU operations
⇨ In early machines, simplicity and reliability determined the overall structure of the CPU and its ALU
– Result was that machines were built around a single register, known as the accumulator
– The accumulator was used in almost all ALU related instructions
⇨ The power and flexibility of the CPU and the ALU is improved through increases in the complexity of the hardware
– Use general register sets to store operands and, addresses, and results
– Increase the capabilities of the ALU
– Use special hardware to support transfer of execution between points in a program
– Replicate functional units within the ALU to permit concurrent operations

⇨ Problem: design a <u>minimal cost</u> yet <u>fully functional</u> ALU
– What building block components would be included?
⇨ Solution:
– Only 2 basic components are required to produce a fully functional ALU

» A bit-wide full adder unit
» A 2-input NAND gate
– NAND is a functionally complete logic operation
– Similarly, if you can add, all other arithmetic operations can be derived from addition.
– To conduct operations on multiple bit words is clearly tedious!
– Goal then is to develop arithmetic and logic circuitry that is algorithmically efficient while remaining cost effective

Integer Representation

⇨ Sign-magnitude format
- o Positional representation using n bits
- o Left most bit position is the sign bit
  - » 0 for positive number
  - » 1 for negative number
- o Remaining n-1 bits represent the magnitude
- o Range: $\{-2^{n-1}-1, +2^{n-1}-1\}$
- o Problems:
  - » Sign must be considered during arithmetic operations
  - » Dual representation of zero (-0 and +0)

⇨ Ones complement format
- o Binary case of diminished radix complement (recall the 9s complement for base 10 numbers?)
- o Negative numbers are represented by a bit-by-bit complementation of the (positive) magnitude (the process of negation)
- o Sign bit interpreted as in sign-magnitude format
- o Examples (8-bit words):
  +42 = 0 00101010
  -42 = 1 11010101
- o Still have a dual representation for zero (all zeros and all ones)

⇨ Twos complement format
- o Binary case of radix complement
- o Negative numbers, -X, are represented by the pseudo-positive number $2^n - |X|$
- o With 2n symbols
  - » 2^{n-1}-1 positive numbers
  - » 2^n-1 negative numbers

- o Given the representation for +X, the representation for -X is found by taking the 1s complement of +X and adding 1
- o Caution: avoid confusion with "2s complement format (representation) and the 2s complement operation
- o Converting between two word lengths (e.g., convert an 8-bit format into a 16-bit format) requires a sign extension:
  - ▪ The sign bit is extended from its current location up to the new location
  - ▪ All bits in the extension take on the value of the old sign bit
    +18= 00010010
    +18= 00000000 00010010
    -18 = 11101110
    -18 = 11111111 11101110

Integer Addition of n-bit numbers

⇨ Use of a single full adder is the simplest hardware
- o Must implement an n-repetition for-loop for an n-bit addition
- o This is lots of overhead for a typical addition

⇨ Use a ripple adder unit instead
- o n full adder units cascaded together
- o In adding X and Y together unit i adds Xi and Yi to produce SUM_i and $CARRY_i$
- o Carry out of each stage is the carry in to the next stage
- o Worst case add time is n times the delay of each unit -- despite the parallel operation of each adder unit -- Order (n) delay
- o With signed numbers, watch out for overflow: when adding 2 positive or 2 negative numbers, overflow has occurred if the result has the opposite sign

⇨ Alternatives to the ripple adder
- o Must allow for the worst case delay in a ripple adder
- o In most cases, carry signals do not propagate through the entire adder
- o Provide additional hardware to detect where carries will occur or when the carry propagation is completed
- o Carry Completion Sensing Adders use additional circuitry to detect the time when all carries are completed
  » Signal control unit that add is finished
  » Essentially an asynchronous device

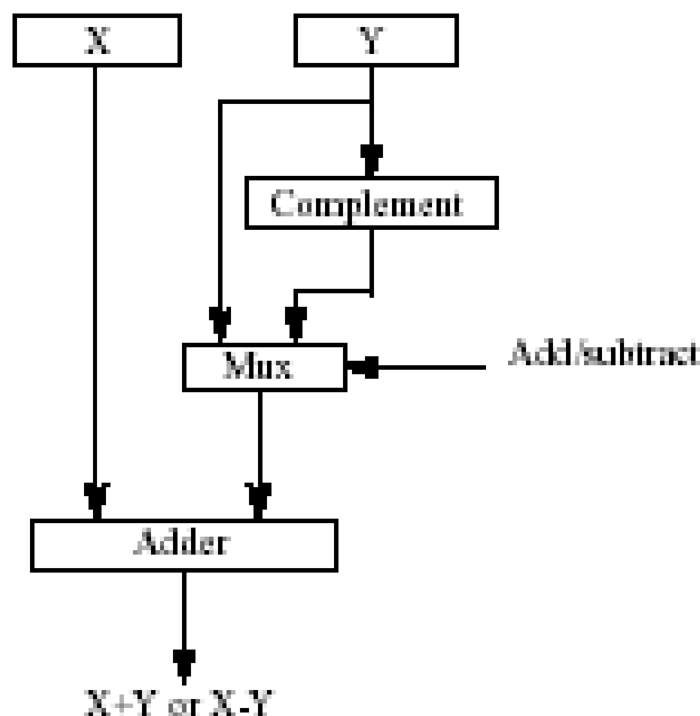
» Typical add times are O(log n)

- Carry Lookahead Adders
  - » Predict in advance what adder stage of a ripple adder will generate a carry out
  - » Use prediction to avoid the carry propagation delays -- generate all of the carries at once
  - » Add time is a constant, regardless of the width, n, of the word -- O(1)
  - » Problem: prediction in stage i requires information from all previous stages -- gates to implement this require large numbers of inputs, making this adder impractical for even moderate values of n

Integer Subtraction

⇨ To perform X-Y, realize that X-Y = X+(-Y)

⇨ Therefore, the following hardware is “typical”



⇨ The other example of addition and substraction

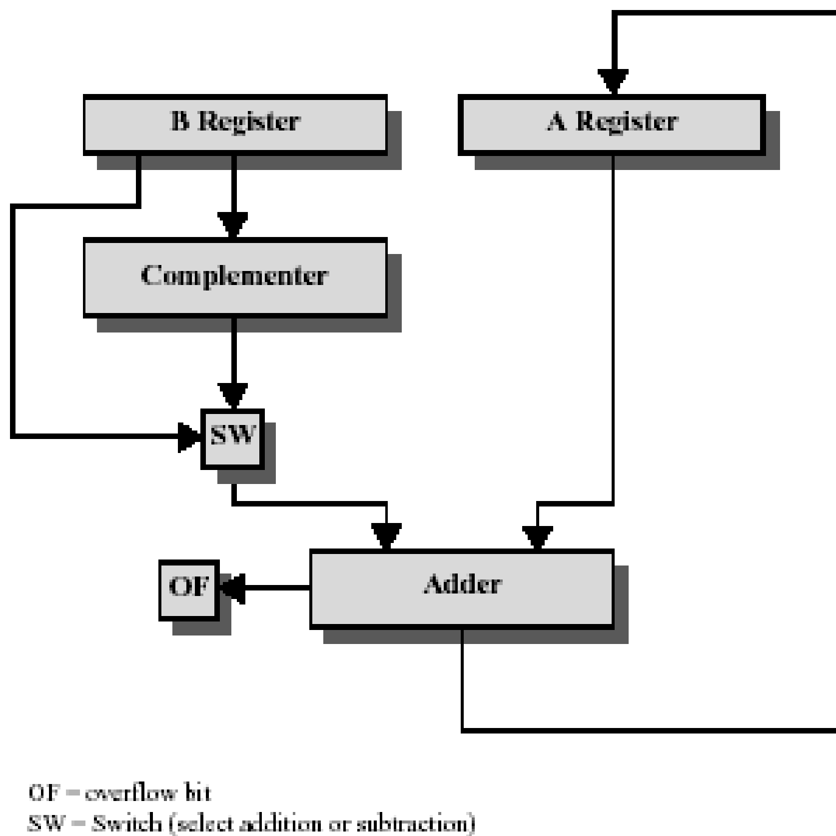


Figure 8.6 Block Diagram of Hardware for Addition and Subtraction

Integer Multiplication

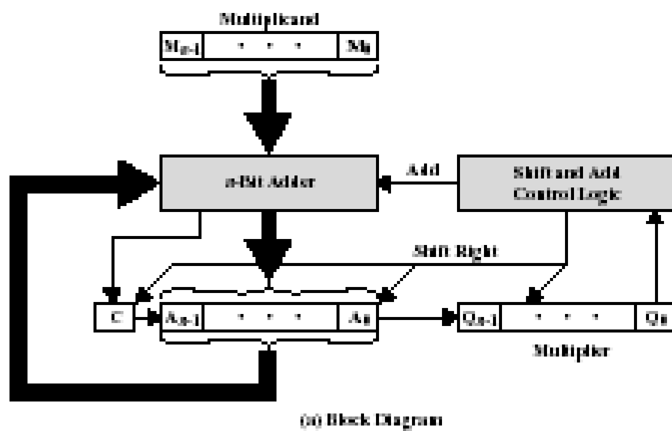
⇨ A number of methods exist to perform integer multiplication
- Repeated addition: add the multiplicand to itself "multiplier" times
- Shift and add -- traditional "pen and paper" way of multiplying (extended to binary format)
- High speed (special purpose) hardware multipliers

⇨ Repeated addition
- Least sophisticated method
- Just use adder over and over again
- If the multiplier is n bits, can have as many as 2^n iterations of addition -- $O(2^n)$!!!!
- Not used in an ALU

⇨ Shift and add
- Computer's version of the pen and paper approach:

```
     1011      (11)
x    1101      (13)
===============
     1011
    00000      Partial products
   101100
  1011000
===============
10001111       (143)
```

- The computer version accumulates the partial products into a running (partial) sum as the algorithm progresses
- Each partial product generation results in an add and shift operation



(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 8.7 (product in A, Q)

Figure 8.8 Hardware Implementation of Unsigned Binary Multiplication

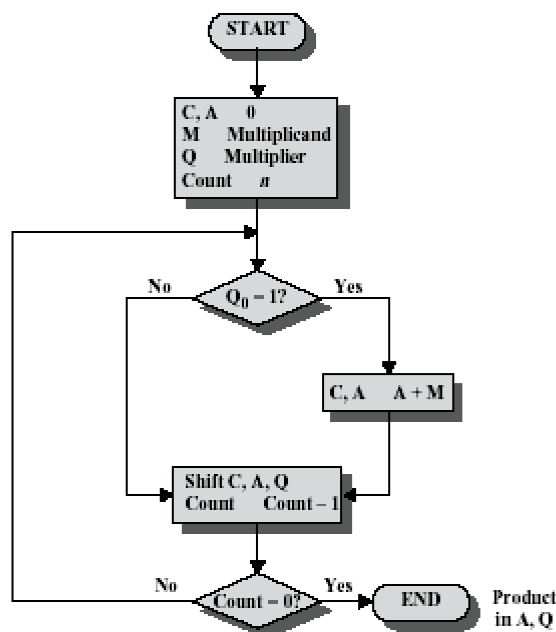


Figure 8.9 Flowchart for Unsigned Binary Multiplication

⇨ To multiply signed numbers (2s complement)
- o Normal shift and add does not work (problem in the basic algorithm of no sign extension to 2n bits)
- o Convert all numbers to their positive magnitudes, multiple, then figure out the correct sign
- o Use a method that works for both positive and negative numbers
  » Booth's algorithm is popular (recoding the multiplier)
- o Booth's algorithm
  - » As in S&A, strings of 0s in the multiplier only require shifting (no addition steps)
  - » "Recode" strings of 1s to permit similar shifting
  - » String of 1s from 2^u down to 2^v is treated as $2^{u+1} - 2^v$
  - » In other words,
    - ▪ At the right end of a string of 1s in the multiplier, perform a subtraction
    - ▪ At the left end of the string perform an addition
    - ▪ For all of the 1s in between, just do shifts
  - » Hardware modifications required in Figure 8.7
    - ▪ Ability to perform subtraction
    - ▪ Ability to perform arithmetic shifting rather than logical shifting (for sign extension)
    - ▪ A flip flop for bit Q_{-1}
  - » To determine operation (add and shift, subtract and shift, shift) examine the bits Q_0Q_{-1}
    - ▪ 00 or 11: just shift
    - ▪ 10: subtract and shift
    - ▪ 01: add and shift

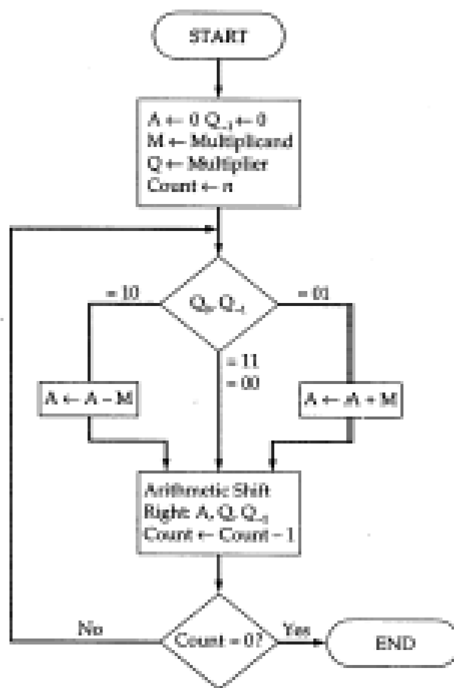
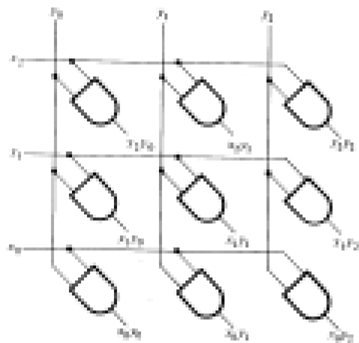


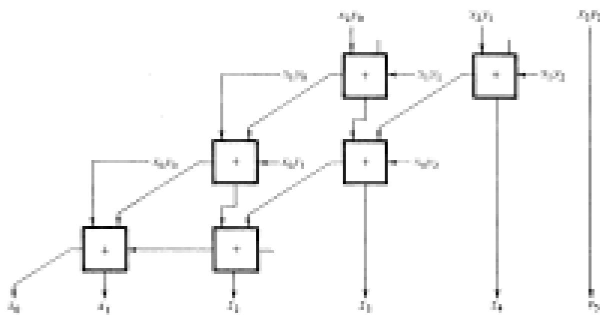
Figure 8.11 Booth's algorithm for multiplication

- » Advantages of Booth:
  - ▪ Treats positive and negative numbers uniformly
  - ▪ Strings of 1s and 0s can be skipped over with shift operations for faster execution time

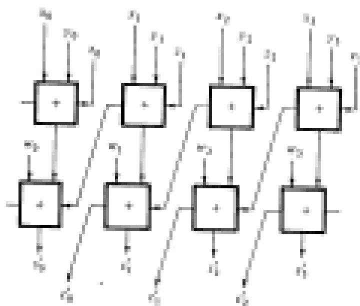
⇨ High performance multipliers

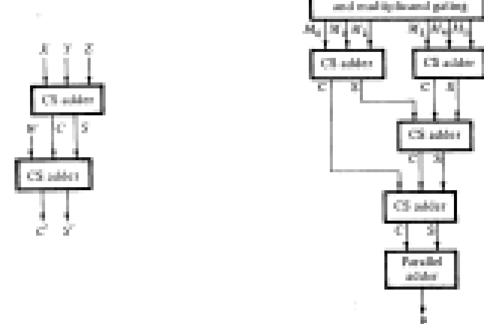
- o Reduce the computation time by employing more hardware than would normally be found in a S&A-type multiplier unit
- o Not generally found in general-purpose processors due to expense
- o Examples
  - » Combinational hardware multipliers
  - » Pipelined Wallace Tree adders from Carry-Save Adder units





Combinational adder circuit [Hay88]





Pipelined Wallace Tree adder [Hay88]

Integer Division

- ⇨ Once you have committed to implementing multiplication, implementing division is a relatively easy next step that utilizes much of the same hardware
- ⇨ Want to find quotient, Q, and remainder, R, such that
  D = Q x V + R
- ⇨ Restoring division for unsigned integers
  - o Algorithm adapted from the traditional "pen and paper" approach
  - o Algorithm is of time complexity O(n) for n-bit dividend
  - o Uses essentially the same ALU hardware as the Booth multiplication algorithm
    - » Adder / subtractor unit
    - » Double wide shift register AQ that can be shifted to the left
    - » Register for the divisor
    - » Control logic

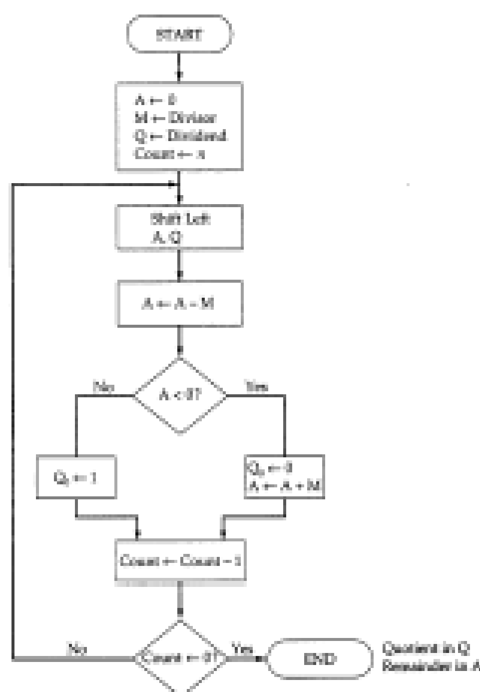


Figure 8.15 Restoring division algorithm for unsigned integers

- ⇨ For two's complement numbers, must deal with the sign extension "problem"
- ⇨ Algorithm:
  - o Load M with divisor, AQ with dividend (using sign bit extension)
  - o Shift AQ left 1 position

- o If M and A have same sign, A<--A-M, otherwise A<--A+M
- o Q0<--1 if sign bit of A has not changed or (A=0 AND Q=0), otherwise Q0=0 and restore *A
- o Repeat shift and +/- operations for all bits in Q
- o Remainder is in A, quotient in Q
  - » If the signs of the divisor and the dividend were the same, quotient is correct, otherwise, Q is the 2's complement of the quotient

A	Q	M = 0011	A	Q	M = 1101
0000	0111	Initial Value	0000	0111	Initial Value
0000	1110	Shift	0000	1110	Shift
1101		Subtract	1101		Add
0000	1110	Restore	0000	1110	Restore
0001	1100	Shift	0001	1100	Shift
1110		Subtract	1110		Add
0001	1100	Restore	0001	1100	Restore
0011	1000	Shift	0011	1000	Shift
0000		Subtract	0000		Add
0000	1001	Set $Q_0 = 1$	0000	1001	Set $Q_0 = 1$
0001	0010	Shift	0001	0010	Shift
1110		Subtract	1110		Add
0001	0010	Restore	0001	0010	Restore

(a) (7) ÷ (3) (b) (7) ÷ (−3)

A	Q	M = 0011	A	Q	M = 1101
1111	1001	Initial Value	1111	1001	Initial Value
1111	0010	Shift	1111	0010	Shift
0010		Add	0010		Subtract
1111	0010	Restore	1111	0010	Restore
1110	0100	Shift	1110	0100	Shift
0001		Add	0001		Subtract
1110	0100	Restore	1110	0100	Restore
1100	1000	Shift	1100	1000	Shift
1111		Add	1111		Subtract
1111	1001	Set $Q_0 = 1$	1111	1001	Set $Q_0 = 1$
1111	0010	Shift	1111	0010	Shift
0010		Add	0010		Subtract
1111	0010	Restore	1111	0010	Restore

(c) (−7) ÷ (3) (d) (−7) ÷ (−3)

Figure 8.16 2's complement division examples

Floating Point Representation

⇨ Integer fixed point schemes do not have the ability to represent very large or very small numbers

⇨ Need the ability to dynamically move the decimal point to a convenient location

⇨ Format: $+/-M \times R^{+/-E}$

⇨ Mantissas are stored in a normalized format
- o Either 1.xxxxx or 0.1xxxxx
- o Since the 1 is required, don't need to explicitly store it in the data word -- insert it for calculations only

⇨ Exponents can be positive or negative values

- Use biasing (Excess coding) to avoid operating on negative exponents
- Bias is added to all exponents to store as positive numbers

⇨ For a fixed n-bit representation length, 2^n combinations of symbols
- If floating point increases the range of numbers in the format (compared to integer representation) then the “spacing” between the numbers must increase
  This causes a decrease in the format’s precision
- If more bits are allocated to the exponent, range is increased at the expense of decreased precision
- Similarly, more mantissa bits increases the precision and reduces the range
- The radix is chosen at design time and is not explicitly represented in the format
  - » Small -- smaller range
  - » Large -- increased range but loss of significant bits as a result of mantissa alignment when normalizing

⇨ Problems to deal with in the format
- Representation of zero
- Over and underflow and how to detect
- Rounding operations

⇨ IEEE 754 format
- Defines single and double precision formats (32 and 64 bits)
- Standardizes formats across many different platforms
- Radix 2
- Single
  - » Range 10^{-38} to 10^{+38}
  - » 8-bit exponent with 127 bias
  - » 23-bit mantissa
- Double
  - » Range 10^{-308} to 10^{+308}
  - » 11-bit exponent with 1023 bias
  - » 52-bit mantissa

⇨ Floating point arithmetic operations
- Addition and subtraction
  - » Align mantissas
  - » Add or subtract mantissas
  - » Post normalize

- o Multiplication
  - » Add exponents
  - » Multiply mantissas
  - » Post normalize
- o Division
  - » Subtract exponents
  - » Divide mantissas
  - » Post normalize

Summary

⇨ Looked at the basicd concepts associated with computer arithmetic
- o Number representation
- o Basic ALU construction
- o Hardware and software implementations of multiplication and division operations
- o Floating point numbers and operations

Pertanyaan Pendalaman Materi

1. Salah bagian dari CPU adalah ALU. Jelaskan prinsip kerja ALU.

2. Ada keterbatasan kemampuan ALU, antara lain bagaimana membuat ALU mempunyai fungsi yang optimal dan harganya murah. Bagaimana komponen ALU dapat dibentuk dengan mempertimbangkan kedua unsure tersebut.

3. Jelaskan bagaimana merepresentasikan bilangan bertanda dalam system bilangan biner.

4. Jelaskan system bilangan biner dalam format “ones complement” maupun “twos complement”.

5. Jelaskan prinsip perkalian dalam system bilangan biner.