**ORAGANISI DAN ARSITEKTUR KOMPUTER**

**COMPUTER ARITHMATIC**



**Daftar Isi**

**Daftar Isi 2**

**Computer Aritmatik 3**

**a. Unit Aritmatika dan Logika 3**

**b. Representasi Bilangan Bulat..............................................................................3**

**c. Aritmatika Bilangan Bulat ................................................................................4**

**d. Representasi Titik Mengambang......................................................................6**

**e. Aritmatika Titik Mengambang........................................................................ 9**

**COMPUTER ARITHMATIC**

**A.Unit Aritmatika dan Logika**

ALU adalah bagian dari komputer yang benar-benar melakukan operasi aritmatika dan logis pada data. Semua elemen lain dari sistem komputer — unit kontrol, register, memori, I / O — ada terutama untuk membawa data ke ALU untuk diproses dan kemudian mengambil kembali hasilnya. Kami memiliki, dalam arti, mencapai inti atau inti dari komputer ketika kita mempertimbangkan ALU.

ALU dan memang, semua komponen elektronik di komputer, didasarkan pada penggunaan perangkat logika digital sederhana yang dapat menyimpan digit biner dan melakukan operasi logika Boolean sederhana.

Gambar 10.1 menunjukkan, secara umum, bagaimana ALU saling berhubungan dengan sisa prosesor. Operan untuk operasi aritmatika dan logika disajikan ke ALU dalam register, dan hasil operasi disimpan dalam register. Register ini adalah lokasi penyimpanan sementara dalam prosesor yang dihubungkan oleh jalur sinyal ke ALU (misalnya, lihat Gambar 2.3). ALU juga dapat menetapkan bendera sebagai hasil dari suatu operasi. Sebagai contoh, sebuah bendera overflow diatur ke 1 jika hasil perhitungan melebihi panjang register yang akan disimpan.

**ALU**

**Contr**

**ol**

**signals**

**Operand**

**r**

**egisters**

**Flags**

**Result**

**r**

**egisters**

Nilai-nilai bendera juga disimpan dalam register di dalam prosesor. Prosesor menyediakan sinyal yang mengontrol pengoperasian ALU dan pergerakan data ke dalam dan keluar dari ALU.

**B.Representasi Bilangan Bulat**

Dalam sistem bilangan biner, bilangan acak dapat direpresentasikan hanya dengan angka nol dan satu, tanda minus (untuk bilangan negatif), dan periode, atau titik radix (untuk angka dengan komponen pecahan).

**-**1101.01012**=**-13.312510

Untuk keperluan penyimpanan dan pemrosesan komputer, kami tidak memiliki manfaat simbol khusus untuk tanda minus dan titik radix. Hanya digit biner (0 dan 1) yang dapat digunakan untuk mewakili angka. Jika kita terbatas pada bilangan bulat non-negatif, representasi itu langsung.

Kata 8-bit dapat mewakili angka dari 0 hingga 255, seperti:

00000000 **=** 0

00000001 **=** 1

00101001 **=** 41

10000000 **=** 128 11111111 **=** 255

Secara umum, jika urutan n-bit dari digit biner an-1an-2 c a1a0 ditafsirkan sebagai integer A unsigned, nilainya adalah

*n***a-**1

*A* **=** 2*iai*

*i***=**0

**C.Aritmatika Bilangan Bulat**

Dalam representasi tanda-besaran, aturan untuk membentuk negasi dari bilangan bulat sederhana: membalikkan bit tanda. Dalam notasi dua melengkapi notasi, negasi dari integer dapat dibentuk dengan aturan berikut:

1. Ambil pelengkap Boolean dari setiap bit integer (termasuk bit tanda). Yaitu, atur masing- masing 1 hingga 0 dan masing-masing 0 hingga 1.

2. Memperlakukan hasil sebagai integer biner unsigned, tambahkan 1.

Proses dua langkah ini disebut sebagai operasi komplemen pasangan, atau pengambilan komplemen pasangan dari bilangan bulat.

|  |  |
| --- | --- |
| **+**18 | **=** 00010010 (twos complement) |
| bitwise complement | **=** 11101101  **+** 1  11101110 **=** **-**18 |

Seperti yang diharapkan, negatif dari angka negatif itu sendiri:

|  |  |
| --- | --- |
| **-**18 | **=** 11101110 (twos complement) |
| bitwise complement | **=** 00010001  **+** 1  00010010 **=** **+**18 |

Kita dapat menunjukkan validitas operasi yang baru saja dijelaskan menggunakan definisi dari representasi komplemen kedua dalam Persamaan (10.2). Sekali lagi, menginterpretasikan urutan n-bit digit biner an-1an-2 c a1a0 sebagai komparator komplemen ganda, sehingga nilainya adalah

*n***a-**2

* + - 1. **= -**2*n***-**1*an***-**1 **+** 2*iai*

*i***=**0

Sekarang bentuk kompilasi bitwise, an-1an-2 c a0, dan, memperlakukan ini sebagai integer unsigned, tambahkan 1. Akhirnya, menginterpretasikan urutan n-bit dari digit biner sebagai dua komposer integer B, sehingga nilainya adalah

*n***a-**2

* + - 1. **= -**2*n***-**1*an***-**1 **+** 1 **+** 2*iai*

*i***=**0

Sekarang, kami ingin A = -B, yang berarti A + B = 0. Ini mudah ditunjukkan sebagai benar:

*n***a-**2 **b**

*A* **+** *B* **= -**(*an***-**1 **+** *an***-**1)2*n***-**1 **+** 1 **+ a** 2*i*(*ai* **+** *ai*) **= -**2*n***-**1 **+** 1 **+a**2*i***b**

*i***=**0

*n***-**2 *i***=**0

**= -**2*n***-**1 **+** 1 **+** (2*n***-**1 **-** 1)

**a**

**= -**2*n***-**1 **+** 2*n***-**1 **=** 0

Derivasi sebelumnya mengasumsikan bahwa pertama-tama kita dapat memperlakukan komplemen bitwise A sebagai integer unsigned untuk tujuan menambahkan 1, dan kemudian memperlakukan hasilnya sebagai integer komplemen pasangan. Ada dua kasus khusus yang perlu dipertimbangkan. Pertama, pertimbangkan A = 0. Dalam hal ini, untuk representasi 8-bit:

|  |  |
| --- | --- |
| 0 | **=** 00000000 (twos complement) |
| bitwise complement | **=** 11111111  **+** 1  100000000 **=** 0 |

Ada yang melakukan posisi bit paling signifikan, yang diabaikan. Hasilnya adalah bahwa negasi 0 adalah 0, sebagaimana seharusnya.

Kasus khusus kedua lebih merupakan masalah. Jika kita mengambil negasi dari pola bit 1 diikuti oleh n - 1 nol, kita mendapatkan kembali angka yang sama. Misalnya, untuk kata-kata 8-bit,

|  |  |
| --- | --- |
| **+**128 | **=** 10000000 (twos complement) |
| bitwise complement | **=** 01111111  **+** 1  10000000 **=** **-**128 |

Beberapa anomali semacam itu tidak dapat dihindari. Jumlah pola bit yang berbeda dalam kata n-bit adalah 2n, yang merupakan bilangan genap. Kami ingin mewakili bilangan bulat positif dan negatif dan 0. Jika jumlah yang sama dari bilangan bulat positif dan negatif diwakili (tanda besarnya), maka ada dua representasi untuk 0. Jika hanya ada satu representasi 0 (berpasangan komplemen), maka ada harus berupa jumlah yang tidak seimbang dari angka positif dan negatif yang diwakili. Dalam kasus pasangan kata komplemen, untuk panjang n-bit, ada representasi untuk - 2n-1 tetapi tidak untuk + 2n-1.

**D.Representasi Titik Mengambang**

Dengan notasi titik-tetap (misalnya, pelengkap dua kata) dimungkinkan untuk mewakili kisaran bilangan bulat positif dan negatif yang berpusat pada atau dekat 0. Dengan mengasumsikan titik biner atau radix tetap, format ini memungkinkan representasi angka dengan komponen pecahan demikian juga.

Pendekatan ini memiliki keterbatasan. Jumlah yang sangat besar tidak dapat diwakili, juga tidak dapat fraksi yang sangat kecil. Selanjutnya, bagian pecahan dari hasil bagi dalam pembagian dua bilangan besar bisa hilang.

Untuk angka desimal, kita dapatkan batasan ini dengan menggunakan notasi ilmiah. Dengan demikian, 976.000.000.000.000 dapat direpresentasikan sebagai 9.76 \* 1014, dan 0.0000000000000976 dapat direpresentasikan sebagai 9.76 \* 10-14, Apa yang telah kami lakukan, pada dasarnya, secara dinamis menggeser titik desimal ke lokasi yang nyaman dan menggunakan eksponen 10 untuk menjaga lacak titik desimal itu. Ini memungkinkan rentang angka yang sangat besar dan sangat kecil diwakili dengan hanya beberapa digit.

Pendekatan yang sama ini dapat diambil dengan bilangan biner. Kami dapat mewakili angka dalam formulir

{S \* B {E

Nomor ini dapat disimpan dalam kata biner dengan tiga bidang:

■ Tanda: plus atau minus

■ Signifikan dan S

■ Eksponen E

**Sign of significand**

**8**

**bits**

**Significand**

**23**

**bits**

**Biased exponent**

**1.1010001** × **210100  = 0 10010011 10100010000000000000000 = 1.6328125** × **220 –1.1010001** × **210100  = 1 10010011 10100010000000000000000 = –1.6328125** × **220**

**1.1010001** × **2–10100 = 0 01101011 10100010000000000000000 = 1.6328125** × **2–20**

**–1.1010001** × **2–10100 = 1 01101011 10100010000000000000000 = –1.6328125** × **2–20**

Format 32-Bit Floating-Point Khas

B dasar adalah implisit dan tidak perlu disimpan karena sama untuk semua nomor. Biasanya, diasumsikan bahwa titik radix berada di sebelah kanan paling kiri, atau paling signifikan, sedikit dari significand. Artinya, ada satu bit di sebelah kiri titik radix.

Prinsip-prinsip yang digunakan dalam merepresentasikan bilangan floating-point biner paling baik dijelaskan dengan sebuah contoh. Gambar 10.18a menunjukkan format floating-point 32-bit yang khas. Bit paling kiri menyimpan tanda nomor (0 = positif, 1 = negatif). Nilai eksponen disimpan dalam 8 bit berikutnya. Representasi yang digunakan dikenal sebagai representasi bias. Nilai tetap, yang disebut bias, dikurangkan dari lapangan untuk mendapatkan nilai eksponen yang benar. Biasanya, bias sama dengan (2k-1 - 1), di mana k adalah jumlah bit dalam eksponen biner. Dalam hal ini, bidang 8-bit menghasilkan angka 0 hingga 255. Dengan bias dari 127 (27 - 1), nilai eksponen yang sebenarnya berada di kisaran -127 hingga +128. Dalam contoh ini, basis diasumsikan 2.

Tabel 10.2 menunjukkan representasi bias untuk bilangan bulat 4-bit. Perhatikan bahwa ketika bit dari representasi bias diperlakukan sebagai bilangan bulat yang tidak ditandatangani, besaran relatif bilangan tidak berubah. Sebagai contoh, baik dalam representasi bias dan unsigned, angka terbesar adalah 1111 dan angka terkecil adalah 0000. Ini tidak benar untuk sign-magnitude atau twos complement representation. Keuntungan dari representasi bias adalah bahwa angka floating-point yang tidak negatif dapat diperlakukan sebagai bilangan bulat untuk tujuan perbandingan.

Bagian terakhir dari kata (23 bit dalam kasus ini) adalah significand. Setiap angka floating-point dapat diekspresikan dalam banyak cara.

The following are equivalent, where the significand is expressed in binary form:

0.110 **\*** 25

110 **\*** 22

0.0110 **\*** 26

Untuk menyederhanakan operasi pada bilangan floating-point, biasanya diperlukan bahwa mereka dinormalkan. Angka yang normal adalah angka di mana digit paling signifikan dari significand adalah nol. Untuk representasi basis 2, angka normal adalah salah satu di mana bit signifikan yang paling signifikan adalah satu. Seperti telah disebutkan, konvensi yang khas adalah bahwa ada satu bit di sebelah kiri titik radix. Jadi, bilangan bukan nol normal adalah satu dalam bentuk

{1.bbb c b \* 2 {E

dimana b adalah digit biner (0 atau 1). Karena bit paling signifikan selalu satu, tidak perlu menyimpan bit ini; melainkan implisit. Dengan demikian, bidang 23-bit digunakan untuk menyimpan significand 24-bit dengan nilai dalam interval setengah terbuka [1, 2). Diberikan nomor yang tidak normal, nomor dapat dinormalisasi dengan menggeser titik radix ke kanan paling kiri 1 bit dan menyesuaikan eksponen yang sesuai.

Gambar 10.18b memberikan beberapa contoh angka yang disimpan dalam format ini. Untuk setiap contoh, di sebelah kiri adalah bilangan biner; di tengah adalah pola bit yang sesuai; di sebelah kanan adalah nilai desimal. Perhatikan fitur-fitur berikut:

■ Tanda disimpan di bagian pertama kata.

■ Bit pertama dari significand benar selalu 1 dan tidak perlu disimpan di bidang significand.

■ Nilai 127 ditambahkan ke eksponen yang benar untuk disimpan di bidang eksponen.

■ Basisnya adalah 2.

Sebagai perbandingan, Gambar 10.19 menunjukkan kisaran angka yang dapat direpresentasikan dalam kata 32-bit. Menggunakan twos melengkapi representasi integer, semua bilangan bulat dari -231 hingga 231-1 dapat diwakili, dengan total 232 nomor yang berbeda. Dengan contoh format floating-point pada Gambar 10.18, rentang angka berikut ini mungkin:

■ Angka negatif antara - (2 - 2-23) \* 2128 dan -2-127

■ Angka positif antara 2-127 dan (2 - 2-23) \* 2128



**Expr**

**essible negative**

**numbers**

**Negative**

**overfl**

**ow**

**P**

**ositive**

**overfl**

**ow**

**Negativ**

**e**

**underfl**

**ow**

**Zer**

**o**

**P**

**ositive**

**underfl**

**ow**

**Expr**

**essible positiv**

**e**

**numbers**

b) Floating-point numbers

(

**Number**

**line**

**0**

Angka Ekspresibel dalam Format 32-Bit Biasa

Lima wilayah pada garis bilangan tidak termasuk dalam rentang ini:

■ Angka negatif kurang dari - (2 - 2-23) \* 2128, yang disebut overflow negatif

■ Angka negatif lebih besar dari 2-127, disebut aliran negatif

■ Nol

■ Angka positif kurang dari 2-127, disebut underflow positif

■ Angka positif lebih besar dari (2 - 2-23) \* 2128, yang disebut luapan positif

Representasi yang disajikan tidak akan mengakomodasi nilai 0. Namun, seperti yang akan kita lihat, representasi floating-point aktual termasuk pola bit khusus untuk menunjuk nol. Overflow terjadi ketika hasil operasi aritmatika dalam nilai absolut yang lebih besar daripada yang dapat diekspresikan dengan eksponen 128 (mis., 2120 \* 2100 = 2220). Underflow terjadi ketika besaran fraksional terlalu kecil (mis., 2-120 \* 2-100 = 2-220). Underflow adalah masalah yang kurang serius karena hasilnya secara umum dapat secara memuaskan didekati oleh 0.

Penting untuk dicatat bahwa kami tidak mewakili lebih banyak nilai individual dengan notasi floating-point. Jumlah maksimum nilai yang berbeda yang dapat diwakili dengan 32 bit masih 232. Apa yang telah kita lakukan adalah menyebarkan angka-angka itu dalam dua rentang, satu positif dan satu negatif. Dalam praktiknya, sebagian besar angka floating-point yang ingin direpresentasikan hanya kira-kira. Namun, untuk bilangan bulat berukuran sedang, representasi itu tepat.

Juga, perhatikan bahwa angka yang diwakili dalam notasi titik-mengambang tidak diberi jarak sepanjang garis bilangan, seperti juga angka-angka titik-tetap. Nilai yang mungkin semakin dekat bersama di dekat asal dan jauh terpisah saat Anda bergerak menjauh, seperti yang ditunjukkan pada Gambar 10.20. Ini adalah salah satu trade-off dari matematika floating-point: Banyak perhitungan menghasilkan hasil yang tidak tepat dan harus dibulatkan ke nilai terdekat yang dapat mewakili notasi.

Dalam jenis format yang digambarkan pada Gambar 10.18, ada trade-off antara rentang dan presisi. Contoh ini menunjukkan 8 bit dikhususkan untuk eksponen dan 23 ke significand. Jika kita meningkatkan jumlah bit dalam eksponen, kita memperluas jangkauan angka yang dapat diukur. Tetapi karena hanya sejumlah nilai yang berbeda yang dapat diekspresikan, kami telah mengurangi kepadatan angka-angka itu dan oleh karena itu ketepatannya. Satu-satunya cara untuk meningkatkan jangkauan dan presisi adalah dengan menggunakan lebih banyak bit. Dengan demikian, sebagian besar komputer menawarkan, setidaknya, nomor presisi tunggal dan nomor putusan ganda. Misalnya, prosesor dapat mendukung format presisi tunggal 64 bit, dan format presisi ganda 128 bit

Jadi ada trade-off antara jumlah bit dalam eksponen dan jumlah bit dalam significand. Tapi itu lebih rumit dari itu. Basis eksponen yang tersirat tidak perlu 2. Arsitektur IBM S / 390, misalnya, menggunakan basis 16 [ANDE67b]. Format ini terdiri dari eksponen 7-bit dan significand 24-bit

***N* 0 *n* 2*n* 4*n***

In the IBM base-16 format,

0.11010001 **\*** 210100 **=** 0.11010001 **\*** 16101

and the exponent is stored to represent 5 rather than 20.

Keuntungan menggunakan eksponen yang lebih besar adalah bahwa rentang yang lebih besar dapat dicapai untuk jumlah bit eksponen yang sama. Tetapi ingat, kami belum meningkatkan jumlah nilai yang berbeda yang dapat diwakili. Dengan demikian, untuk format tetap, basis eksponen yang lebih besar memberikan rentang yang lebih besar dengan mengorbankan presisi yang kurang.

**E.Aritmatika Titik Mengambang**

merangkum operasi dasar untuk aritmatika floating-point. Untuk penambahan dan pengurangan, perlu untuk memastikan bahwa kedua operan memiliki nilai eksponen yang sama. Ini mungkin membutuhkan menggeser titik radix pada salah satu operan untuk mencapai keselarasan. Perkalian dan pembagian lebih mudah. Operasi floating-point dapat menghasilkan salah satu dari kondisi ini:

■ Exponent overflow: Eksponen positif melebihi nilai eksponen maksimum yang mungkin. Dalam beberapa sistem, ini dapat ditetapkan sebagai + ∞ or - ∞.

■ Kelebihan eksponen: Eksponen negatif kurang dari nilai eksponen minimum yang mungkin (mis., - 200 kurang dari -127). Ini berarti bahwa angka tersebut terlalu kecil untuk diwakili, dan dapat dilaporkan sebagai 0.

Nomor Titik Mengambang dan Operasi Aritmatika

|  |  |
| --- | --- |
| **Floating-Point Numbers** | **Arithmetic Operations** |
| *X* **=** *XS* **\*** *BXE Y* **=** *YS* **\*** *BYE* | *X* **+** *Y* **=** (*XS* **\*** *BXEE***-***YEE* **+***EYS*) **\*** *BYEE***f***XE* **…** *YE*  *X* **-** *Y* **=** (*XS* **\*** *BX* **-***Y* **-** *YS*) **\*** *BY*  *X* **\*** *Y* **=** (*XS* **\*** *YS*) **\*** *BX* **+***YE*   1. **a***XS***b** *XE***-***YE*   **= \*** *B*   1. *YS* |

Examples:

1. **=** 0.3 **\*** 102 **=** 30
2. **=** 0.2 **\*** 103 **=** 200

*X* **+** *Y* **=** (0.3 **\*** 102**-**3 **+** 0.2) **\*** 103 **=** 0.23 **\*** 103 **=** 230

*X* **-** *Y* **=** (0.3 **\*** 102**-**3 **-** 0.2) **\*** 103 **=** ( **-** 0.17) **\*** 103 **= -**170

*X* **\*** *Y* **=** (0.3 **\*** 0.2) **\*** 102**+**3 **=** 0.06 **\*** 105 **=** 6000

*X* **,** *Y* **=** (0.3 **,** 0.2) **\*** 102**-**3 **=** 1.5 **\*** 10**-**1 **=** 0.15

■ Significand underflow: Dalam proses penyelarasan, digit dapat mengalir dari ujung kanan significand. Seperti yang akan kita diskusikan, beberapa bentuk pembulatan diperlukan.

■ Signifikan & overflow: Penambahan dua tanda signifikan yang sama dapat menghasilkan sebagian besar yang signifikan. Ini bisa diperbaiki dengan penataan kembali, seperti yang akan kami jelaskan.

**EE201: Digital Circuits and Systems**

**Section 3 – Arithmetic Logic Unit**



3.1 ALU



Definition

* Key processing element of a microprocessor that performs arithmetic and logic operations

Description

* Directed by Control Unit, ALU performs operations such as ADD, SUB, NOT, OR, AND, XOR
* Data is inputted from and outputted to the Register Array
* Control Signals from Control Unit determine what type of operation is performed
* Input data consists of two operands: operand A and operand B stored in registers and having n bits
* Output data consists of result S
* ALU also outputs Status Signals such as:
  + Zero (when the result of the operation is 0)
  + Negative (when the operation result is < 0)
  + Carry (when the operation results in carry)
  + Overflow (when the result exceeds the number of bits allocated for its storage)
  + Etc.

3.2 Addition

3.2.1 Ripple Carry Adder (RCA)

RCA Goal:

RCA Cell:





RCA Cell Minimisations and Implementation:Sum Logic







|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bi** | **Ai** | **Ci-1** | **Si** | **Ci** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **0** |
| **0** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** | **1** |
| **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **0** | **1** |
| **1** | **1** | **0** | **0** | **1** |
| **1** | **1** | **1** | **1** | **1** |







Carry Logic



RCA Delay:

Using d as delay per carry bit, an n-bit RCA has worst case delay of…

TD = n \* d

d will depend on technology used, see e.g.

Example

An 8-bit RCA Adder is implemented using 2-input NAND technology. The delay of each NAND gate is given as 500 pS. After what time will a valid result be visible at the output?

NAND Logic

Use NAND gates to derive other logic gates and functions…

AND GATE



OR GATE



Each Carry Bit will pass through 4 500pS NAND gates

d=4x500 = 2000pS

Adder delay: 2000 \* n = 2000 \* 8

=> 16000 pS or 16nS

3.2.2 Carry Look-ahead Adder (CLA)

CLA Rationale:

* RCA sequentially computes carry bits and based on them addition result bits, causing long delays
* Delay can be reduced by examining all inputs simultaneously and producing the carry bits for each next stage

CLA Principle:

We notice that a carry is produced in stage i if:

* Stage i generates a carry
* Stages i-1 generates a carry and stage i propagates it
* Stage i-2 generates a carry and stages i and i-1 propagate it
* Etc.

A carry is generated only if both Ai and Bi are 1:

* Gi = Ai Bi

A carry is propagated if carry is not generated and at least one of Ai and Bi is 1:

* Pi = !Ai Bi + Ai !Bi

Carry for stage i:

* Ci = Gi + Pi Ci-1

CLA Cell:

CLA Cell Implementation:







4-Bit CLA Carry Implementation:





C0 = G0 + P0 C-1

C1 = G1 + P1 C0

= G1 + P1 (G0 + P0 C-1)

= G1 + P1G0 + P1P0 C-1

C2 = G2 + P2 C1

= G2 + P2 (G1 + P1G0 + P1P0 C-1)

= G2 + P2G1 + P2P1G0 + P2P1P0 C-1

C3 = G3 + P3 C2

=G3 +P3 (G2 + P2G1 + P2P1G0 + P2P1P0 C-1)

= G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0 C-1

CLA Advantages:

* Any Carry Bit, Ci requires only 2 gate levels.
* To produce Gi or Pi from Ai and Bi => 1 level
* Parallelizes operation to improve time.
* Lower delay => faster result

CLA Disadvantages:

* Above C3, equations get very complex.
* Gates with high number of inputs slower.
* Additional Logic => Larger Area, More Power

Accumulator

Rationale:

* Many calculations consist of repeated additions and subtractions performed on the result of the previous operation.
* Accumulator (ACC) is dedicated register used to perform these repeated operations.



Usage:

1. Initialise ACC with base value from memory.
2. Each instruction will then either add or subtract from ACC base value.
3. When finished, Store ACC value to Data Memory.

3.3 Multiplication

3.3.1 Binary Multiplication

Method:

* Multiplication can be performed by treating the multiplier unit as a combinational circuit, multiplicand and multiplier bits as inputs and the bits of the result as outputs
* In general multiplies positive numbers only, however can be extended for any numbers.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **A1** | **A0** | **B1** | **B0** | **P3** | **P2** | **P1** | **P0** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **0** | **0** | **0** | **0** |
| **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **1** | **0** | **1** | **0** | **0** | **0** | **1** |
| **0** | **1** | **1** | **0** | **0** | **0** | **1** | **0** |
| **0** | **1** | **1** | **1** | **0** | **0** | **1** | **1** |
| **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **0** | **0** | **1** | **0** | **0** |
| **1** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
| **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| **1** | **1** | **0** | **1** | **0** | **0** | **1** | **1** |
| **1** | **1** | **1** | **0** | **0** | **1** | **1** | **0** |
| **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** |





Homework: minimisation of P1, P2, P3 and implementation using NAND gates!



`

3.3.2 “Add & Shift” Multiplication

Principle:

* Multiplication is usually performed by computers by repeating additions
* The principle is based on the “pencil and paper” method that requires the computation of partial results and shifting them before they are added in order to calculate the final result

“Pencil and Paper” Method:

* Multiplies positive numbers only
* E.g. 1510 \* 1310 = 11112 \* 11012



* Result verification: 110000112 = 19510

Possible Machine Method:

* Multiplies positive numbers only
* E.g. 1510 \* 1310 = 11112 \* 11012

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Multiplicand (Y)** | | | | |  | **1** | **1** | **1** | **1** | **(+1510)** |
| **Multiplier (X)** | | | | |  | **1** | **1** | **0** | **1** | **(+1310)** |
| **Result (R)** | | | | | | | | | | **Action** |
| **Step** | **C** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **Initialisation** |
|  |  | **1** | **1** | **1** | **1** |  |  |  |  | **1 => ADD (R, Y)** |
|  |  | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **Right Shift** |
| **➀** |  | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |  |
|  |  | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0 => NOP** |
| **➁** |  | **0** | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **Right Shift** |
|  |  | **1** | **1** | **1** | **1** |  |  |  |  | **1 => ADD (R, Y)** |
| Carry | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **0** | **0** | **Right Shift** |
| **➂** |  | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **0** |  |
|  |  | **1** | **1** | **1** | **1** |  |  |  |  | **1 => ADD (R, Y)** |
| **Carry** | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **Right Shift** |
| **➃** |  | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **1** | **Result** |

* Result verification: 110000112 = 19510

Hardware for the Machine Method:

Requires:

* Binary adder to add multiplicand to the shifter partial product if multiplier digit is 1
* Shift register to shift partial product to the right
* Shift register to shift multiplier to the right in order to parse all its digits

Observation:

* If the Multiplicand has Md bits and Multiplier has Mp bits, the result will have Md+Mp bits
* The result will be stored in a Md+Mp bit register and will be initialised with 0s
* As repeated additions and shifts are performed on partial results, the result register can be the accumulator (A)
* As the Multiplicand is added to the left-most Md bits of A and A shifts its content Mp times to the right, the right-most Mp bits of A can store the Multiplier
* As A shifts its content to the right, the Multiplier bits are also shifted, enabling its bit that decides whether to add the Multiplicand to the partial result to be always the last bit in A
* This solution saves a register to store and shift the Multiplier



Question: What signal do you think is missing from the diagram?

3.3.3 Multiplication of Negative Numbers

Rationale:

* The already presented multiplication methods deal with positive numbers only
* There is a need to find a solution to multiply numbers regardless of their sign

Sign-Magnitude Representation:

* Includes the positive number in binary (Magnitude) and a sign-bit that can indicate:
  + 0 - positive number
  + 1 - negative number

Principle:

* The product magnitude is the result of the product of the magnitudes of the multiplicand and the multiplier
* The sign of the product (Psb) is:
  + Positive if the sign of the multiplier (Xsb) and multiplicand (Ysb) are the same
  + Negative if their signs are different

Implementation:



Disadvantage:

* As numbers are represented in two-complement, they require conversion into sign-magnitude, prior to multiplication and then back to two-complement
* The largest negative numbers cannot be multiplied

3.3.4 Booth Algorithm for Multiplication [2’s comp]

Principle:

* Performs additions and subtractions of the Multiplicand, based on the value of the Multiplier bits
* The algorithm looks at two adjacent bits in the Multiplier in order to decide the operation to be performed
* The Multiplier bits are considered from the least significant bit (right-most) to the most significant bit; by default a 0 will be considered at the right of the least significant bit of the Multiplier
* If Multiplicand has Md bits and Multiplier has Mp bits, the result will be stored in a Md+Mp bit register and will be initialised with 0s
* As repeated operations and shifts are performed on partial results, the result register is the accumulator (A)

Algorithm:

1. Initialise the result register with 0; this will store both partial products and the final result
2. If the Multiplier’s bits to be tested are “10”, SUBTRACT the Multiplicand from the partial product
3. If the Multiplier’s bits to be tested are “01”, ADD the Multiplicand from the partial product
4. If the Multiplier’s bits to be tested are “00” or “11”, DO NOTHING
5. Arithmetically RIGHT SHIFT the partial product
6. Sense the next set of adjacent bits of the Multiplier
7. If there are still Multiplier bits to be sensed continue the algorithm at b)

Example:

* E.g. +1110 \* -310 = 010112 \* 111012
* Result verification: 11110111112 = -3310

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Multiplicand (Y)** | | | | |  | **0** | **1** | **0** | **1** | **1** | **(+1110)** |
| **Multiplier (X)** | | | | |  | **1** | **1** | **1** | **0** | **1** | **0 (-310)** |
| **Result (R)** | | | | | | | | | | | **Action** |
| **Step** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **Initialisation** |
|  | **0** | **1** | **0** | **1** | **1** |  |  |  |  |  | **10 => SUB (R, Y)** |
|  | **1** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **A. Right Shift** |
| **➀** | **1** | **1** | **0** | **1** | **0** | **1** | **0** | **0** | **0** | **0** |  |
|  | **0** | **1** | **0** | **1** | **1** |  |  |  |  |  | **01 => ADD (R, Y)** |
|  | **0** | **0** | **1** | **0** | **1** | **1** | **0** | **0** | **0** | **0** | **A. Right Shift** |
| **➁** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **0** | **0** | **0** |  |
|  | **0** | **1** | **0** | **1** | **1** |  |  |  |  |  | **10 => SUB (R, Y)** |
|  | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **A. Right Shift** |
| **➂** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** |  |
|  | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **11 => NOP** |
| **➃** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **A. Right Shift** |
|  | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **0** | **11 => NOP** |
| **➄** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **A. Right Shift** |

Homework: Build the block-level hardware diagram.

3.4 Division

3.4.1 Combinational Circuit-based Division

Homework: Build the combinational circuit that divides 4-bit positive numbers by 2-bit positive numbers

Hint: See Binary multiplication using combinational circuits.

3.4.2 “Subtract and Shift” Division

Algorithm:

1. Initialise the result register (Accumulator - A) with 0; this will store both the Remainder and the Quotient resulted after division
2. Copy Dividend in the least significant part of A
3. If the most significant part of the partial result (AH) is greater or equal to the Divisor, SUBTRACT the Divisor from AH and set the next Quotient bit to 1
4. If the most significant part of the partial result (AH) is smaller than the Divisor, DO NOTHING and set the next Quotient bit to 0
5. LEFT SHIFT the partial result, introducing in the least significant position the Quotient bit just determined
6. If we have performed this algorithm less times than the number of bits the Divisor has, continue the algorithm at b)

Example:

* + E.g. 3510 / 710 = 1000112 / 0001112

Dividend (A): 100011 (3510)

Divisor (B): 000111 (710)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Dividend (A) / Remainder (R) Quotient (Q)** | | | | | | | | | | | | **Action** |
| **AH / Remainder** | | | | | | **AL / Q** | | | | | |
| **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **Left Shift** |
| **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **AH<B => Qo=0** |
| **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **Nop, Left Shift** |
| **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **AH<B => Qo=0** |
| **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **Nop, Left Shift** |
| **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **AH<B => Qo=0** |
| **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **Nop, Left Shift** |
| **0** | **0** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **0** | **AH≥B => Qo=1** |
| **0** | **0** | **0** | **1** | **1** | **1** |  |  |  |  |  |  | **SUB(AH, B)** |
| **0** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **Left Shift** |
| **0** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **AH<B => Qo=0** |
| **0** | **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **Nop, Left Shift** |
| **0** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **AH≥B => Qo=1** |
| **0** | **0** | **0** | **1** | **1** | **1** |  |  |  |  |  |  | **SUB(AH, B)** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **Left Shift** |
| **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **1** |  |

Quotient (Q): 000101 (510)

Remainder (R): 000000 (010)

3.4.3 Multiplication with Reciprocal



Hint: Computation of the Reciprocal can be performed using combinational circuits.

3.5 Operations with Floating Point Binary Numbers

3.5.1 Floating Point Representation

Principle:

* Any real number can be represented in the following form:



* Where A – mantissa and a – exponent
* Floating point representation requires that the floating point binary numbers are normalised

Normalisation:

* A normalised two-complement number has the following requirements:

 or

 or



Example:

|  |  |  |
| --- | --- | --- |
| Decimal | Binary | Floating Point |
| 7 | 0111 | 0.111\*20011 |
| -7 | 1001 | 1.001\*20011 |
| 7/32 | 0.00111 | 0.111\*21110 |

* Non-normalised numbers:

0.001\*20011 101.11\*21011

3.5.2 Floating Point Addition

Addition Algorithm:

* Lets assume that A\*2a and B\*2b will be added
* In order for the addition to be performed, the exponents of the two numbers have to be equalised and their mantissas shifted accordingly:
  + If a>b, B is right shifted a-b places, obtaining B’
  + If a<b, A is right shifted b-a places, obtaining A’
  + If a=b, no shift is required
* The mantissa of the result is obtained:
  + If a>b by adding A and B’
  + If a<b by adding A’ and B
  + If a=b by adding A and B
* The exponent of the result is:
  + If a>b is a
  + If a<b is b
  + If a=b is either a or b
* Normalisation of the result may be needed

Example:

0.100\*20001 +1.000\*20011

a=0001, b=0011;

a<b => A right shifted b-a = 2 places

C = A’ +B = 0.001 + 1.000 = 1.001; c = b

Result: C\*2c = 1.001\*20011

Normalisation Algorithm:

* The mantissa has to be transformed such as it is either
  + Between 0.100000 and 0.111111 for positive numbers

or

* + Between 1.011111 and 1.000000 for negative numbers
* If the result C\*2c is written as CnCn-1…C1C0.C-1C-2…, then C is above range and mantissa is repeatedly right shifted one place at a time and the exponent is incremented each time by one until C0 ⊕C-1= 1
* If the result C\*2c can be written as C0 . C-1C-2…, and C0 ⊕C-1= 0, then C is below the range and mantissa is repeatedly left shifted one place at a time and the exponent is decremented each time by one until C0 ⊕C-1= 1

Example 1:

1010.100\*20100

c=0101, C=1010.100

C right shifted 3 places

c incremented by 1 three times

Result: C\*2c = 1.010100\*20111

Example 2:

0.00100\*20110

c=0110, C=0.00100

C left shifted 2 places

c decremented by 1 two times

Result: C\*2c = 0.10000\*20100

3.5.3 Floating Point Multiplication

Multiplication Algorithm:

* Lets assume that A\*2a and B\*2b will be multiplied
* The multiplication result will have:
  + Mantissa resulted from the multiplication of the two mantissas (C = A\*B)
  + Exponent resulted from the addition of the two exponents (c = a+b)

Example:

0.100\*20010 \*0.101\*20011

a=0010, b=0011;

A=0.100, B=0.101;

C = A \* B = 0.010100

c = a + b = 0101

Normalisation: left shift C 1 place and decrement c by 1

Result: 0.101\*20100

Overflow:

* The result becomes so large that the exponent cannot be represented in the number of bits allocated

3.5.4 Floating Point Division

Division Algorithm:

* Lets assume that A\*2a will be divided by B\*2b
* The division result will have:
  + Mantissa resulted from the division of the two mantissas (C = A/B)
  + Exponent resulted from the subtraction of the two exponents (c = a-b)

Example:

0.101\*20100 /0.100\*20010

a=0100, b=0010;

A=0.101, B=0.100;

C = A / B = 01.010

c = a - b = 0010

Normalisation: right shift C 1 place and increment c by 1

Result: 0.101\*20011

Underflow:

* The result becomes so small that it cannot be represented accurately in the number of bits allocated

FROM :

[www.eeng.dcu.ie/~ee201/03\_Arithmetic\_Logic\_Unit.doc](http://www.eeng.dcu.ie/~ee201/03_Arithmetic_Logic_Unit.doc)