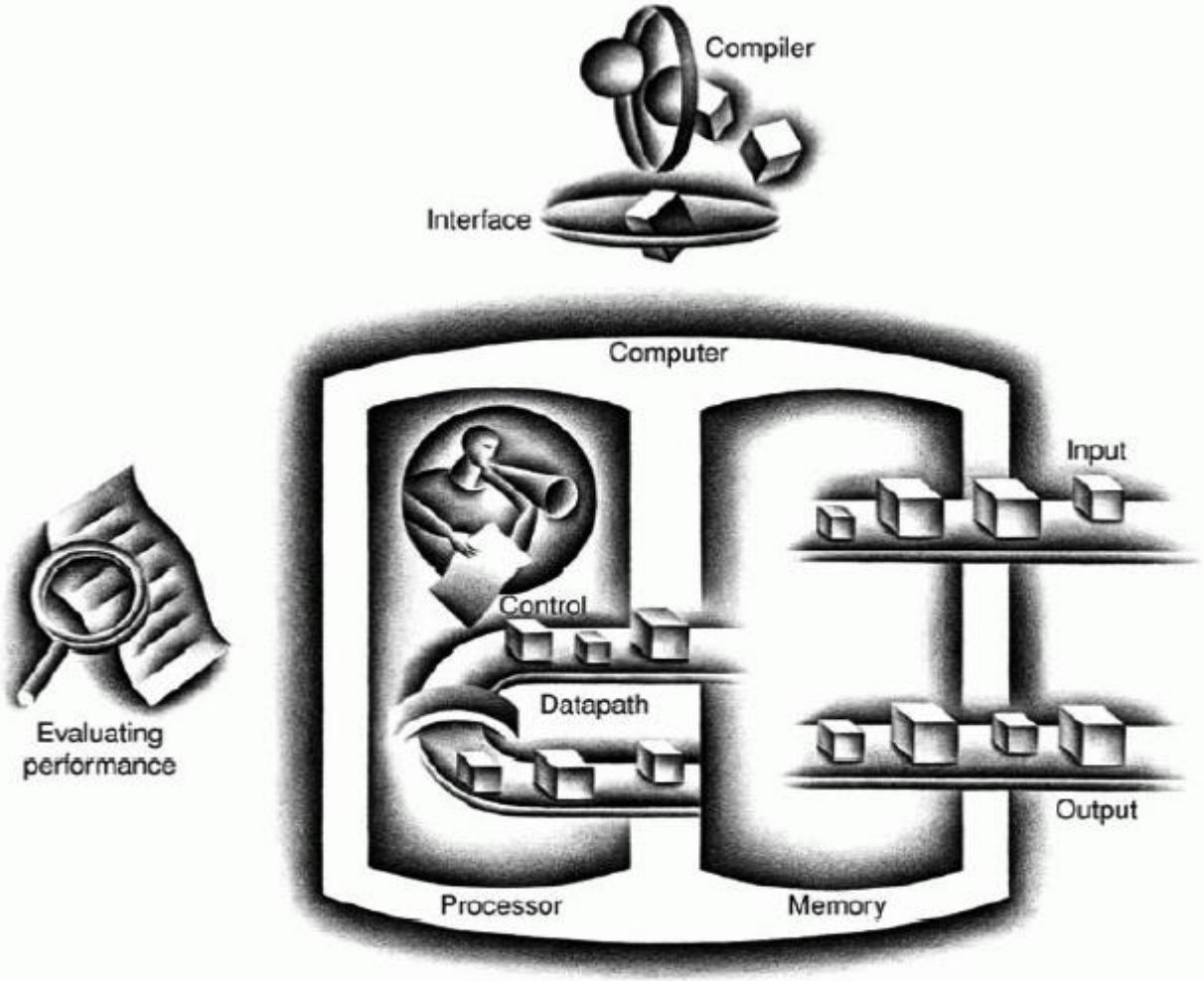
3. Bölüm
BİLGİSAYAR ARİTMETİĞİ



İşaretli ve işaretsiz sayılar

Sayılar herhangi bir taban kullanılarak temsil edilebilir. Günlük hayatımızda kullandığımız sayı sistemi 10'luk (Desimal) sayı sistemiyken bilgisayar 2'lik (Binary)sayı sistemini kullanmaktadır. Herhangi bir tabanda bir sayının oluşumu aşağıdaki gibidir. A_i basamak rakamı, T taban, i basamak değerini ifade eder. Ai basamak rakamı, 0, 1..T-1'e kadarki ardışıl rakamlardan herhangibiri olabilir.

$$(\text{Sayı})_T = \sum_{i=0}^{i=n-1} A_i \cdot T^i + \sum_{j=-m}^{j=-1} A_j \cdot T^j$$

(Tamsayı kısmı) + (kesirli kısmı)

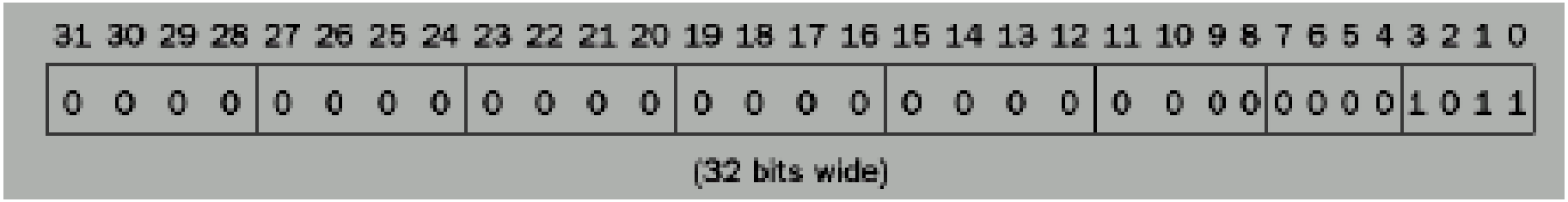
Soru: Bilgisayarlarda niye 2 tabanlı sayı sistemi kullanılır?

Cevap: Bu tabanda kullanılan rakamlar 0 ve 1'dir. Bu rakamların elektriksel işaretler olarak tanınması çok kolaydır.(Transistörün anahtarlama modunda çalıştırılması ile)

1011_{two} sayısının desimal sayı sistemindeki karşılığını hesaplayalım;

$$
\begin{aligned}
&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten} \\
&= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten} \\
&= 8 + 0 + 2 + 1_{ten} \\
&= 11_{ten}
\end{aligned}
$$

Bu sayıyı ikilik tabanda 32 bit ile ifade edecek olursak yeni sayı;



32 bitlik işaretsiz sayılarda yazılabilecek en küçük sayı;

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$$

En büyük sayı;

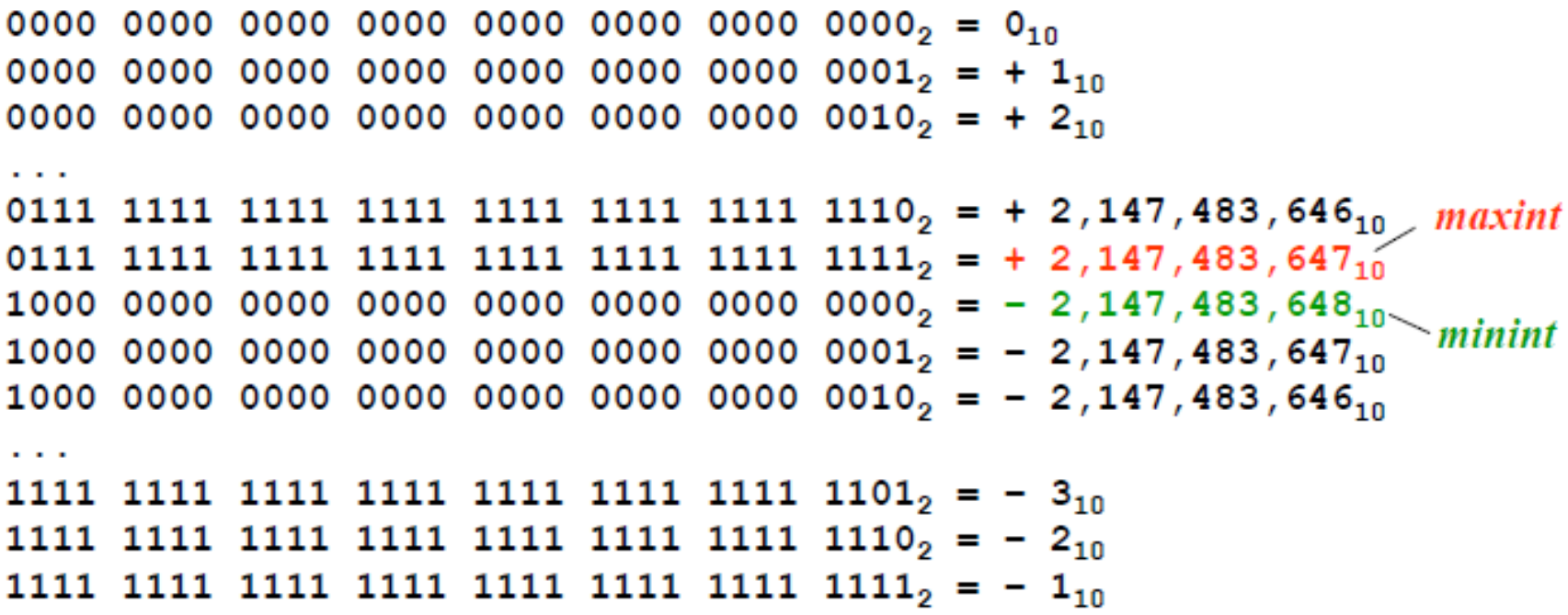
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$$

Bu sayıların desimal karşılıklarının hesaplanması;

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

İşaretli sayılarda ikilik tabanda diğer sayı tabanlarında olduğu gibi ± işaretleri, sayının pozitif ya da negatif olduğunu belirlemez ikilik tabanda işaretli bir sayının işaretini en anlamlı biti (MSB) vermektedir. En anlamlı bit işareti, diğer bitler ise sayıyı ifade etmektedir.

En anlamlı bit ‘0’ ise sayı pozitiftir ve desimal karşılığı işaretsiz sayılarda olduğu gibi hesaplanır. En anlamlı bit ‘1’ ise sayı negatiftir ve desimal karşılığını hesaplayabilmek için 2’ye tümleyeninin alınması gerekmektedir.



Yukarıda 32 bitlik bazı işaretli sayılar ve desimal karşılıkları verilmektedir.

32 bitlik işaretli sayılar ile ifade edilebilecek;

En büyük sayı (2^{31}-1) 2,147,483,647

En küçük sayı (-2^{31}) -2,147,483,648 dir.

- İkilik tabanda verilen işaretli bir sayının desimal karşılığı aşağıdaki şekilde hesaplanmaktadır.

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Buradaki -2^{31} işaret bitini temsil etmektedir.

Örnek: 5 bitlik işaretli sayı için

$(+13)_{10} = 0\ 1101 = 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 8 + 4 + 1 = +13$

$(-13)_{10} = 1\ 0011 = -(1x2^4) + (0x2^3 + 0x2^2 + 1x2^1 + 1x2^0) = -16 + 3 = -13$

İkilik tabandan onluk tabana dönüşüm işlemi

Verilen 32 bitlik işaretli sayının desimal karşılığını bulalım.

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$$

Sayının tüm bitlerini formülde yerine koyarsak;

$$\begin{aligned}
&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0}) \\
&= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0 \\
&= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten} \\
&= -4_{ten}
\end{aligned}$$

MIPS

- **`lb, lbu`**
  - **(lb - load byte) 32 bitlik işaretli sayının en anlamlı 24 bitini işarete göre doldurur.**
  - **(lbu- load byte unsigned) işaretsiz sayılarda anlamlı bitlere 0 atar.**
- **`lh, lhu`**
  - **(lh - Load half) işaretli sayılarda en anlamlı 16 biti işarete göre doldurur.**
  - **(lhu - Load half word unsigned) işaretsiz sayılarda**
- **`slt & slti`**
  - **(slt - set on less than) işaretli sayılarda karşılaştırma**
  - **(slti - set on less than immediate) işaretli tamsayılarda karşılaştırma**
- **`sltu & sltiu`**
  - **(sltu - set on less than unsigned) işaretsiz sayılarda karşılaştırma**
  - **(sltiu -set on less than immediate unsigned) işaretsiz tamsayılarda karşılaştırma**

İşaretli ve işaretsiz sayıların karşılaştırılması

$s0 register'ında aşağıdaki sayı olsun.

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$$

$s1 register'ında aşağıdaki sayı olsun.

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$$

Aşağıdaki tanımlamaya göre $t0 ve $t1 register'larının değerleri ne olur?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

Cevap;

$s1 değeri 1'dir. $s0 işaretli bit tamsayı ise değeri -1, işaretsiz bir tamsayı ise değeri 4294967295 tir.

slt $t0, $s0, $s1 $t0=1 (-1<1)

sltu $t1, $s0, $s1 $t1=0 (4294967295>1)

İşaret değiştirme

- 2 sayısını 32 bitlik -2'ye çeviriniz.

$$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$$

Bir sayının negatifi alınacağı zaman öncelikle tüm bitlerinin tersi alınır, ardından 1 ile toplanarak karşılığı bulunur.

$$
\begin{array}{rl}
 & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1101_{two} \\
+ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\
\hline
= & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1110_{two} \\
= & -2_{ten}
\end{array}
$$

Aynı işlem negatif bir sayının pozitif karşılığını bulmak için de kullanılır.

$$
\begin{array}{rl}
 & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} \\
+ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\
\hline
= & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0010_{two} \\
= & 2_{ten}
\end{array}
$$

İkilik tabanda işaretli 16 bitlik 2'yi 32 bit ile ifade edelim.

$$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

İkilik tabanda işaretli 16 bitlik -2'yi 32 bit ile ifade edelim.

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_{two} \\ + \qquad\qquad\qquad\qquad 1_{two} \\ \hline \end{array}$$

$$= 1111\ 1111\ 1111\ 1110_{two}$$

16 bitlik -2'yi 32 bitlik sayıya çevireceğimiz zaman anlamlı bitlerine '0' değil '1' eklememiz gerekmektedir.

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$

MIPS operands

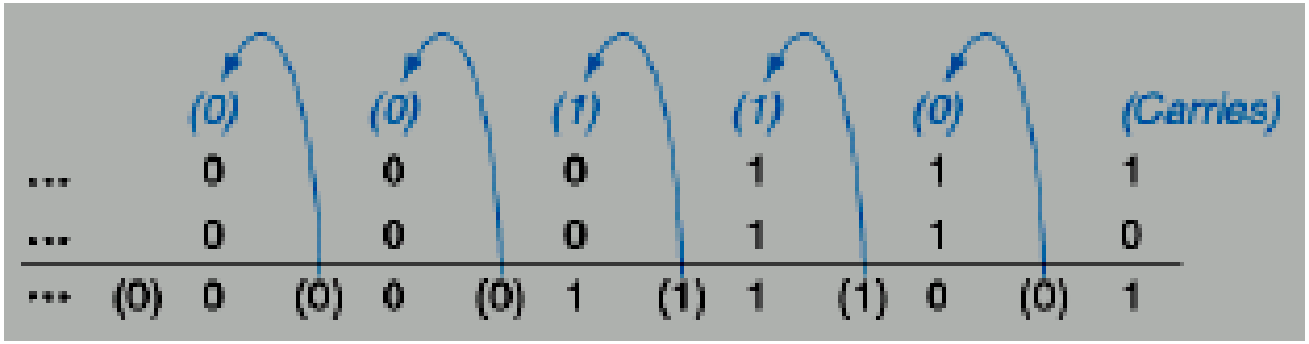
Name	Example	Comments
32 registers	`$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load half unsigned	`lhu $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Halfword memory to register
	store half	`sh $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Halfword register to memory
	load byte unsigned	`lbu $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1,100`	`$s1 = 100 * 2^16`	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1 = ~ ($s2 |$s3)`	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; unsigned numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare < constant; unsigned numbers
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Toplama

6 ile 7 sayılarının ikilik sayı tabanında toplanması;



Basamaklar sağdan sola elde biti de göz önüne alınarak eklenerek soldaki basamağa geçer.

$$\begin{array}{lll} & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\ + & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{two} & = 6_{ten} \\ \hline = & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,1101_{two} & = 13_{ten} \end{array}$$

Çıkarma

Aynı örnek üzerinde çıkarma işlemini yapalım;

$$
\begin{array}{ll}
 & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} = 7_{ten} \\
- & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{two} = 6_{ten} \\
\hline
= & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} = 1_{ten}
\end{array}
$$

Çıkarma işleminde yukarıdaki gibi doğrudan çıkarma yapılabileceği gibi çıkarılan sayının 2'ye tümleyeni alınarak toplamada yapılabilir.

$$
\begin{array}{ll}
 & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} = 7_{ten} \\
+ & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1010_{two} = -6_{ten} \\
\hline
= & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} = 1_{ten}
\end{array}
$$

MIPS

- **`Add, addu`**
  - İşaretli sayılarda toplama
  - İşaretsiz sayılarda toplama
- **`Addi, addiu`**
  - İşaretli sayılarda bir tamsayı ile toplama
  - İşaretsiz sayılarda
- **`Sub, subu`**
  - İşaretli sayılarda çıkarma
  - İşaretsiz sayılarda çıkarma
  - MIPS istisna nedeninin talimat adresini içeren EPC(exception program counter) olarak adlandırılan bir register içerir. sistem kontrolünden ayrılma komutu(MFC0-move from system control) EPCyi genel amaçlı bir register içine kopyalamak için kullanılır.

Toplama ve çıkarma için overflow durumları;

Operation	Operand A	Operand B	Aşağıdaki sonuçlar da taşma oluşur
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

- **Bir işlem sonucunda elde edilen sayısal değer, sözcük uzunluğu ile tanımlanmış değerlerden büyükse, taşma (overflow) vardır denir.**
- **32 bitlik 2 sayı toplandığında/çıkarıldığında sonuç 33 bit ise;**
- **İki pozitif sayı toplandığında sonuç negatif ise (sonucun İşaret biti 1 ise);**
- **İki negatif sayı toplandığında sonuç pozitif ise (sonucun işaret biti 0 ise);**
- **Bir pozitiften bir negatif değer çıkartıldığında sonuç negatif ise (sonucun işaret biti 1 ise)**
- **Bir negatiften bir pozitif değer çıkartıldığında sonuç pozitif ise (sonucun işaret biti 0 ise)**

  overflow (taşma) veya underflow (borç alma) oluştuğu anlaşılır.
- **Bir negatif sayı bir pozitif sayıya eklendiğinde hiçbir zaman taşma olmaz.**
- **Çıkarılan sayılar aynı işaretli ise taşma olmaz.**

Taşma oluşumunun tanınması

– ***add, addi veya sub komutlarının sebep olduğu bir taşma interrupt 'ı (kesme – exception-istisna) oluşur.***

•Kontrol, önceden tanımlanmış adresteki interrupt rutinine atlar.

•Interrupted addresi, mümkün yeniden başlatamalar için kaydedilir.

• MIPS **$epc (exception program counter) registeri içerir.** Bu register interrupt' a sebep olan komutun adresini tutar.
• Move from system control (Sistem kontrolundan ayrılma) komutu ile;

mfco $t0, $epc # copy $epcto register

$epc reg'in içeriği $t0 reg'e aktarılır.

• İşaretsiz sayılarda oluşan (**addu, addiu, subu komutları ile**) overflow'lardaki interrup'lar sürekli olarak sezilmek istenmez. Bu durumdaki taşma yok sayılır. Çünkü işaretsiz sayılarla işlemler genellikle ana hafıza adreslemeleri içindir.

Tavsiye:

QtSPIM'de : EPC ve ilgili reg'leri incele.

MIPS overflowu yakalayabilir, ancak birçok bilgisayarda overflow testi için dallanma şartı bulunmamaktadır. Overflow'u yakalayabilmek için MIPS kodu aşağıdaki gibi yazılır;

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
so no overflow
xor  $t3, $t0, $t1 # signs =; sign of sum match too?
                   # $t3 negative if sum sign different
slt  $t3, $t3, $zero # $t3 = 1 if sum sign different
bne  $t3, $zero, Overflow # All three signs ≠; go to
overflow
```

İşaretsiz sayılarda toplama için ($t0= $t1+$t2) overflow'u yakalayabilmek için yazılan MIPS kodu;

```
addu $t0, $t1, $t2 # $t0 = sum
nor  $t3, $t1, $zero # $t3 = NOT $t1
          # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2 # (2^32 - $t1 - 1) < $t2
          #⟹ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1 < $t1 + $t2) go to
overflow
```

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 - $s3	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	`addu $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	`subu $s1,$s2,$s3`	$s1 = $s2 - $s3	Three operands; overflow undetected
	add immediate unsigned	`addiu $s1,$s2,100`	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	`mfc0 $s1,$epc`	$s1 = $epc	Used to copy Exception PC plus other special registers
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	`lhu $s1,100($s2)`	$s1 = Memory($s2 + 100)	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory($s2 + 100) = $s1	Halfword register to memory
	load byte unsigned	`lbu $s1,100($s2)`	$s1 = Memory($s2 + 100)	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory($s2 + 100) = $s1	Byte from register to memory
	load upper immediate	`lui $s1,100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsign	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Çarpma

Multiplicand	1000
Multiplier	x 1001
	1000
	0000
	0000
	1000
Product	**01001000**

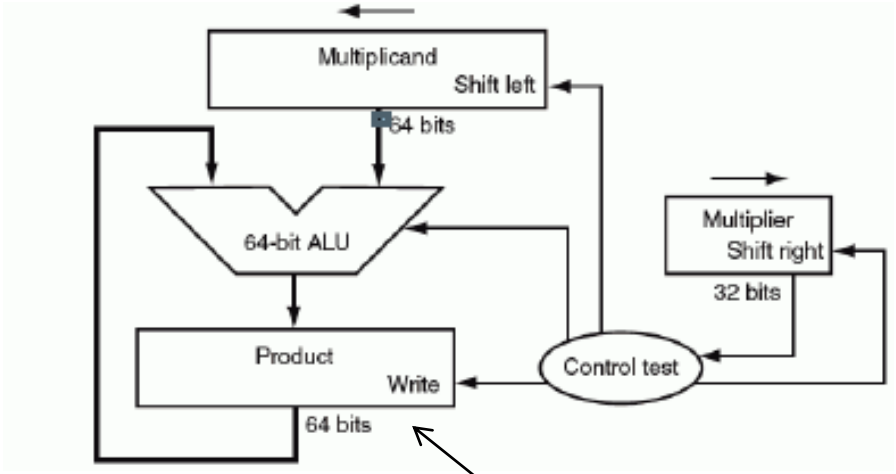
m bit x n bit = m+n bit çarpım (product)

Binary 2 sayıyı çarpma işlemi;

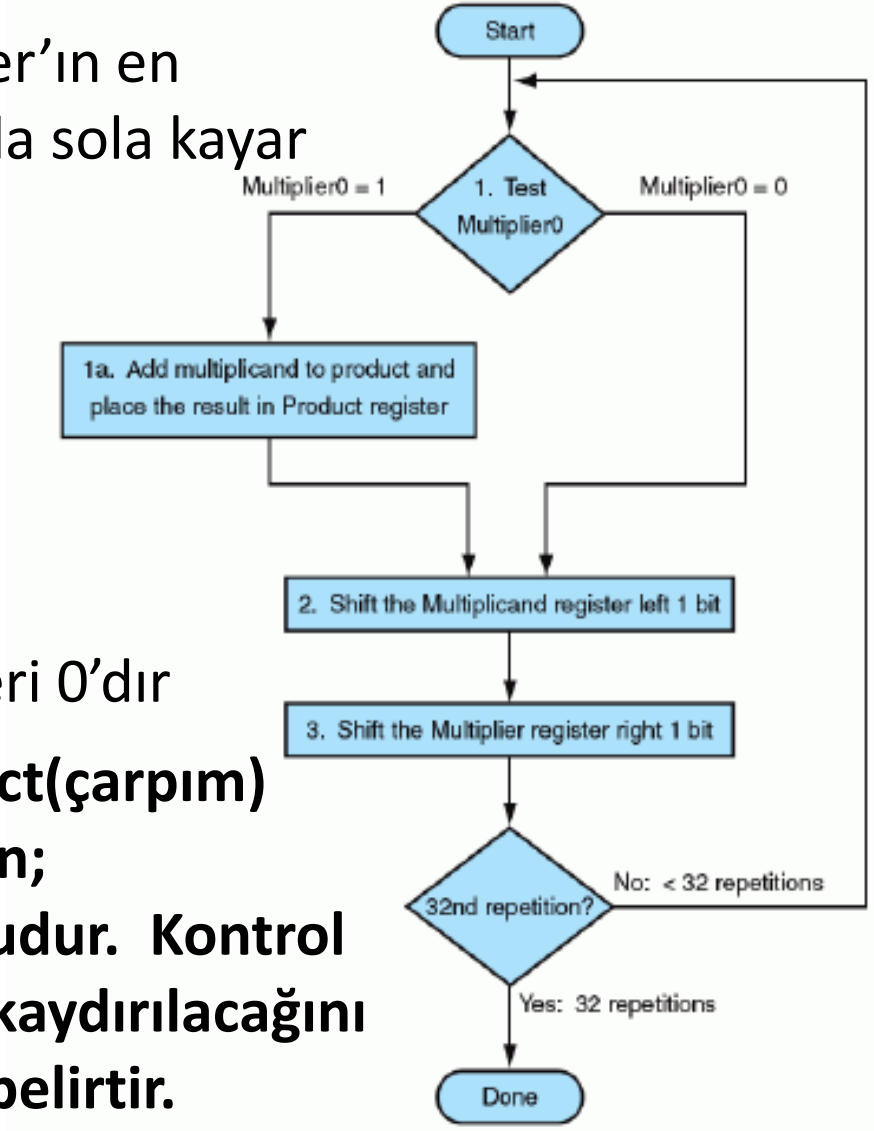
- çarpan(multiplier) bit **1 =>** çarpılanı kopyala (1 x multiplicand)
- çarpan bit **0 =>** yerine 0 koy (0 x multiplicand)

ÇARPMA İŞLEMİNİN DONANIMININ İLK VERSİYONU VE AKIŞ ŞEMASI

32-bit multiplicand(çarpılan) register'ın en anlamsız bitlerindedir ve her adımda sola kayar

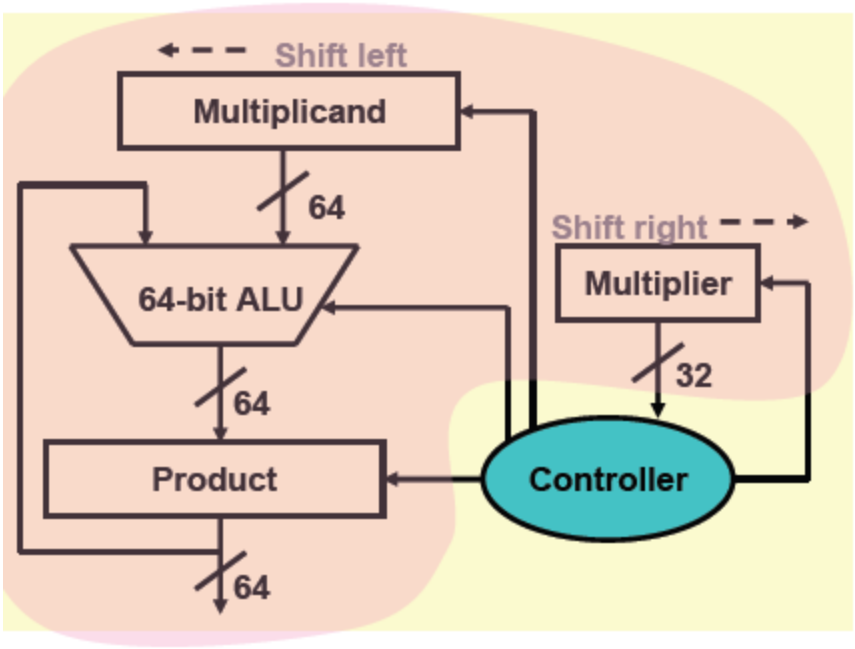


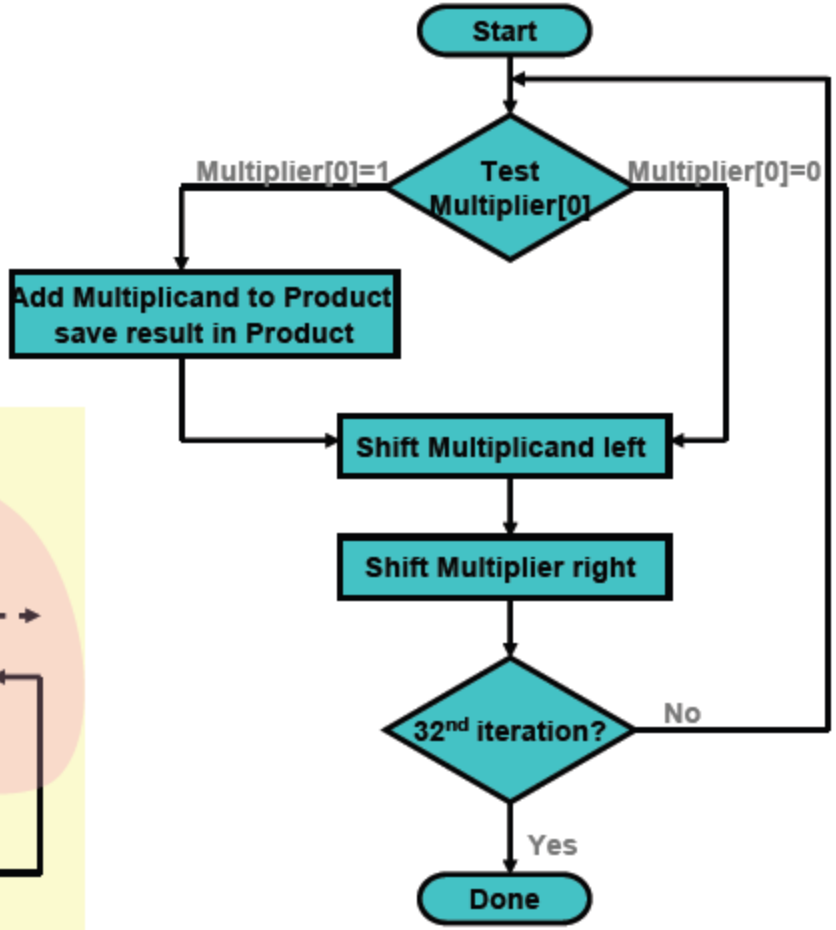
Çarpımın başlangıç değeri 0'dır



Multiplicand(çarpılan) register, product(çarpım) register ve ALU 64-bit genişliğindeyken; multiplier(çarpan) ise 32-bit uzunlukludur. Kontrol bloğu çarpılan ile çarpanın ne zaman kaydırılacağını ve sonucun ne zaman yenileneceğini belirtir.

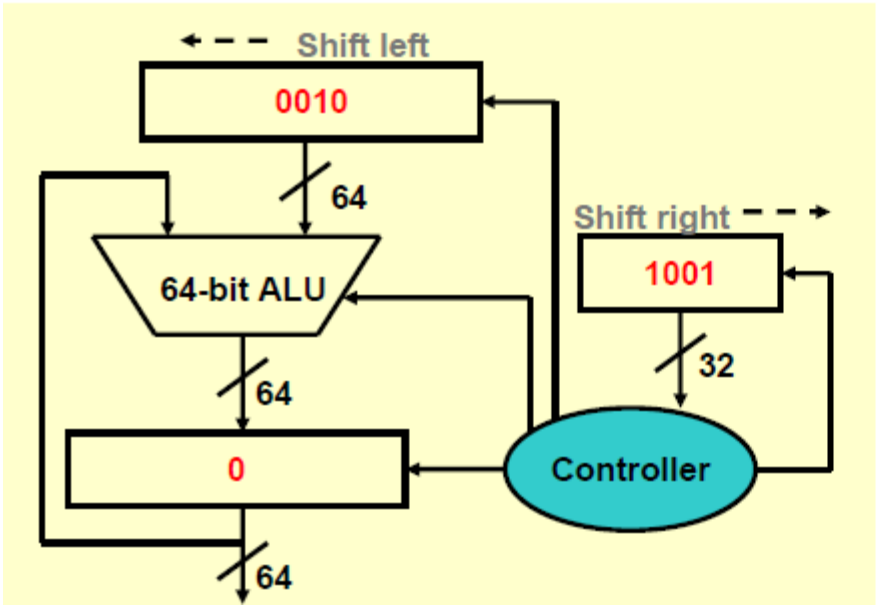
-Çarpma işlemi, bir çok kez kaydırma ve 64 bit toplama işlemiyle başarılır.
-Basit fakat yavaş çalışır.
- Bu işlemi DATAPATh'te hangi parça yapar?

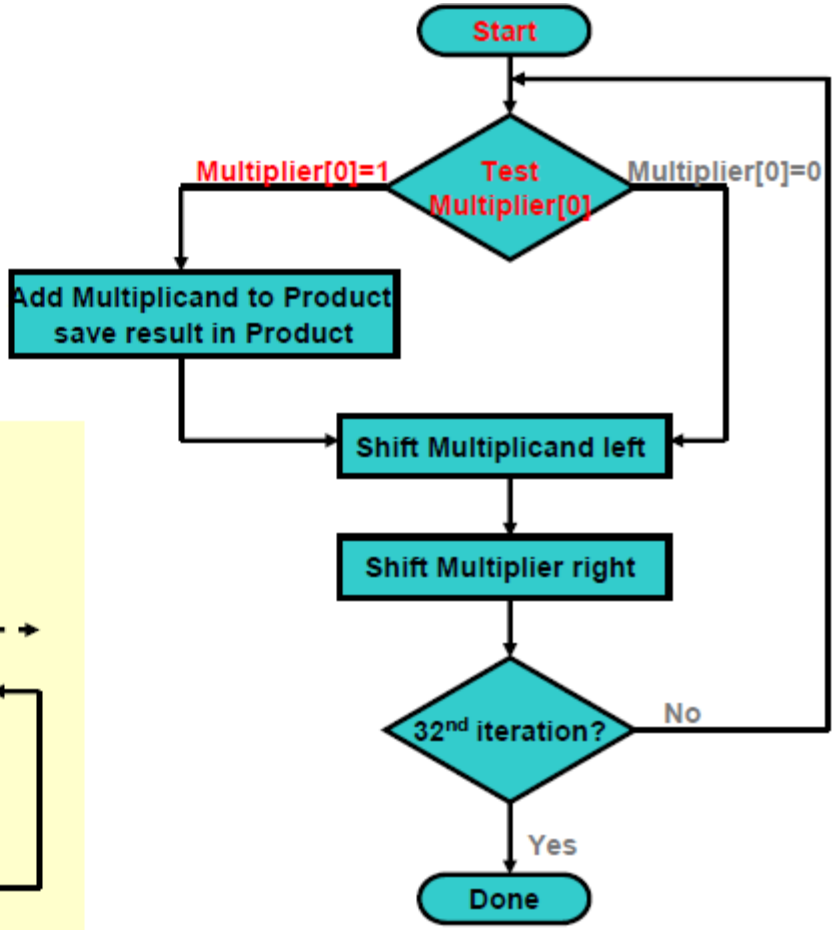




Control Flow

```
    0010   (multiplicand)
  x 1001   (multiplier)
    0010
   0000
  0000
+ 0010
  0010010  (product)
```

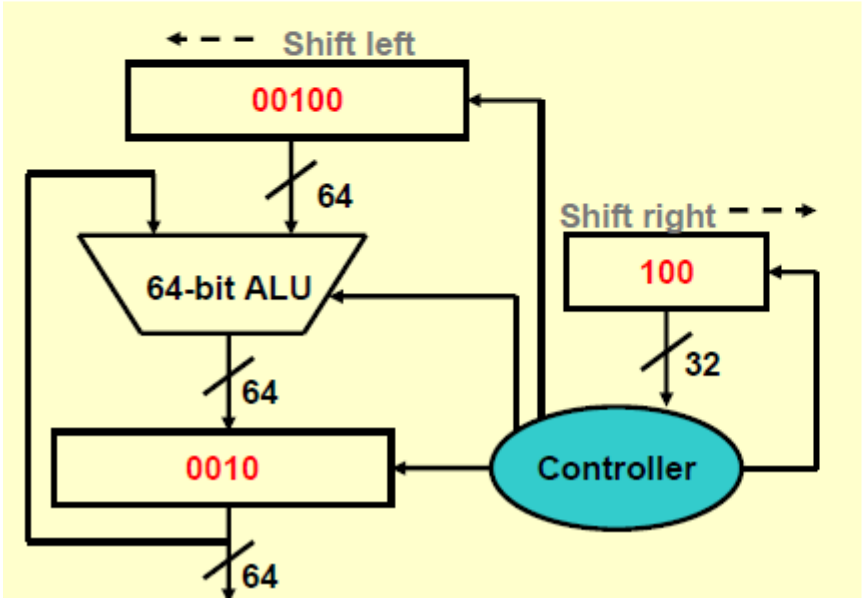


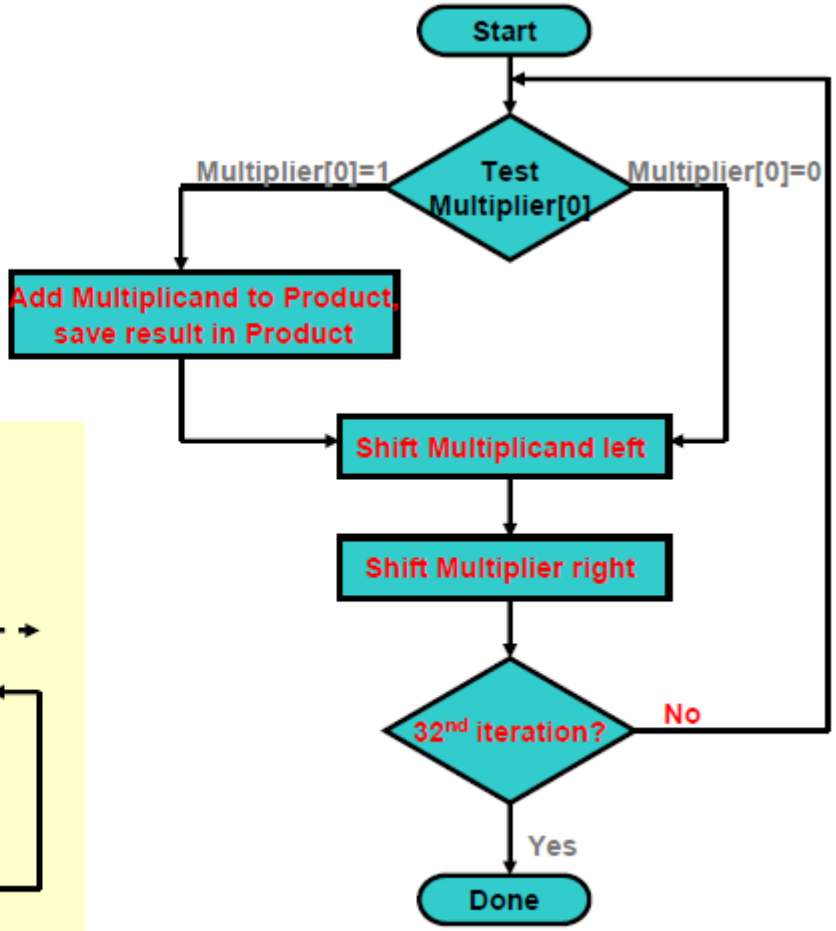


Control Flow

(Ροή Μονάδας Ελέγχου)

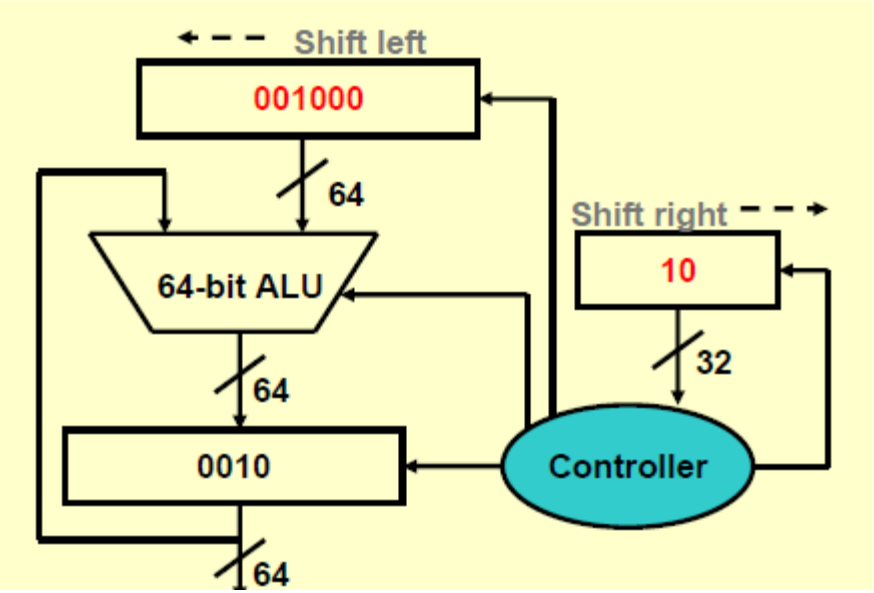
```
    0010   (multiplicand)
  x 1001   (multiplier)
  ------
    0010
   0000
  0000
+ 0010
  -------
  0010010  (product)
```

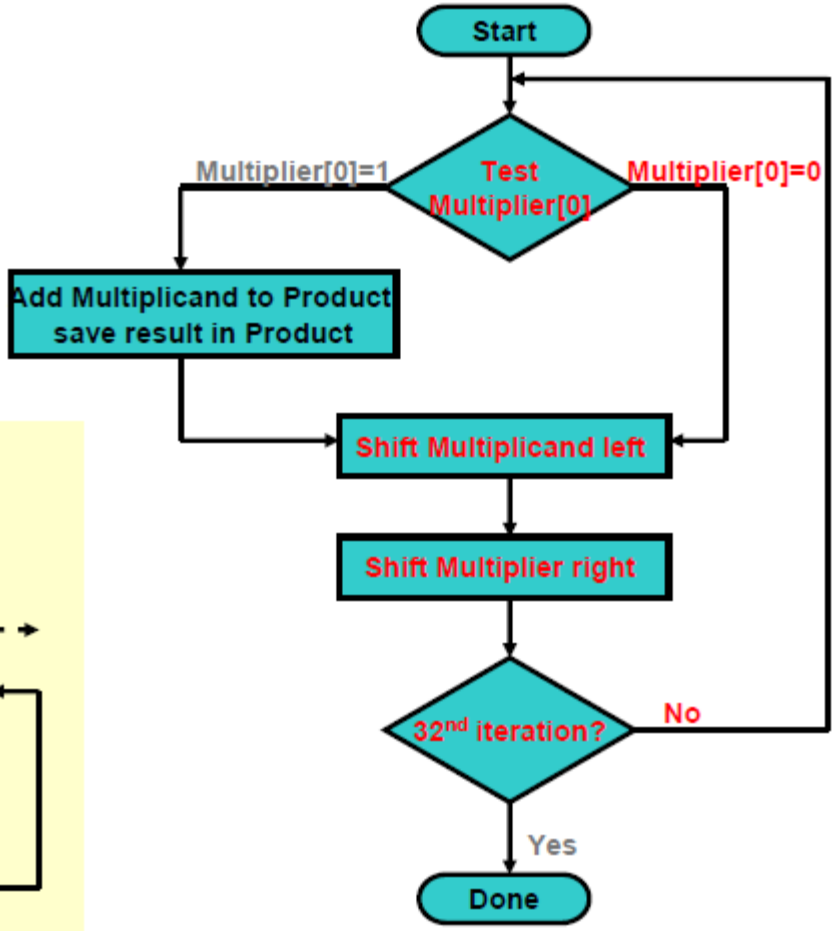




Control Flow

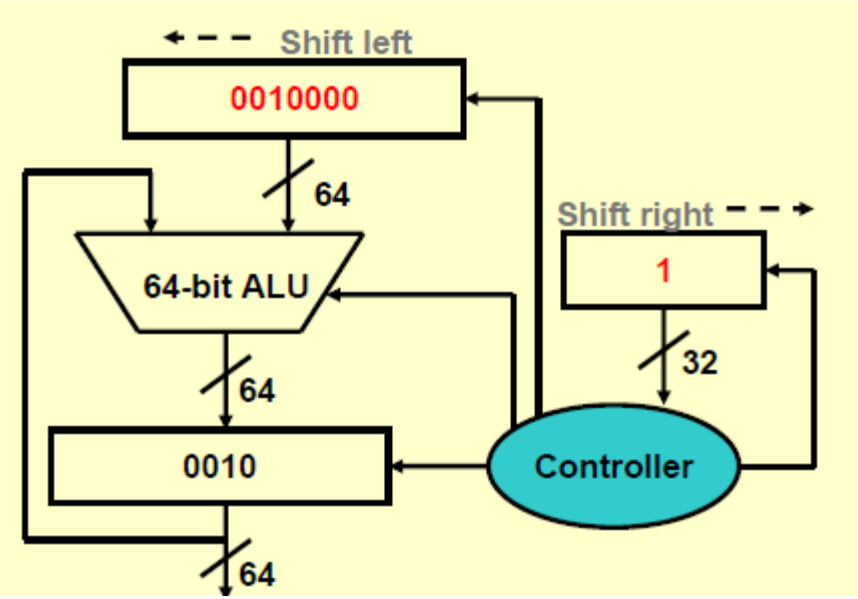
```
    0010   (multiplicand)
  x 1001   (multiplier)
  ------
    0010
   0000
  0000
+ 0010
  -------
  0010010  (product)
```

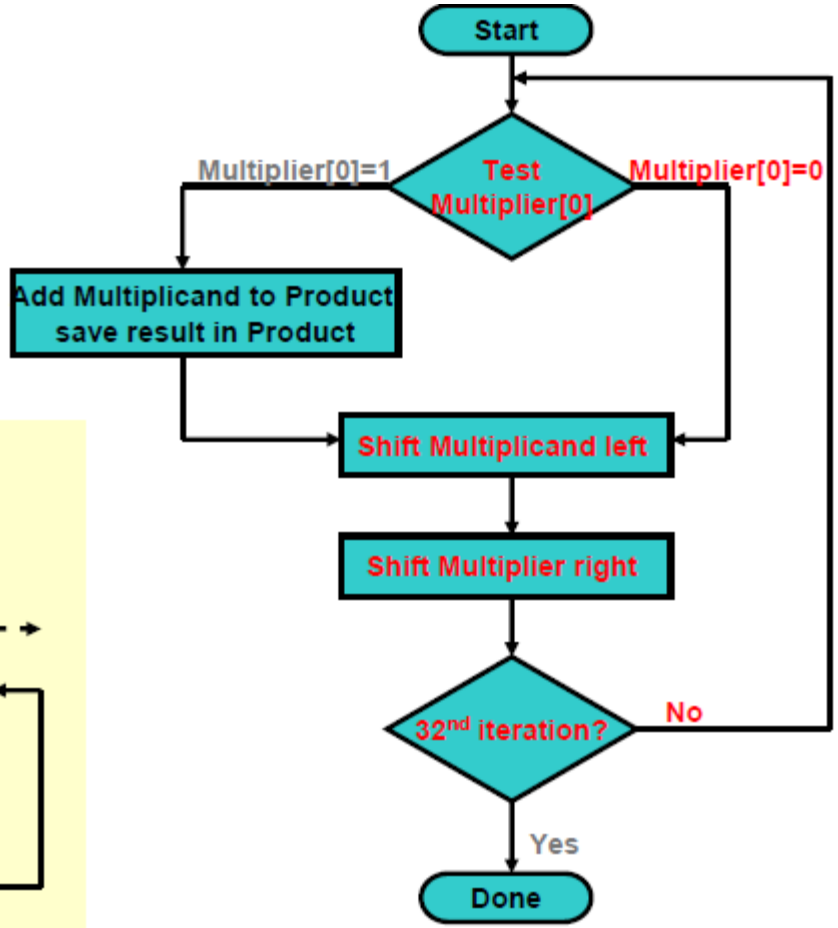




Control Flow

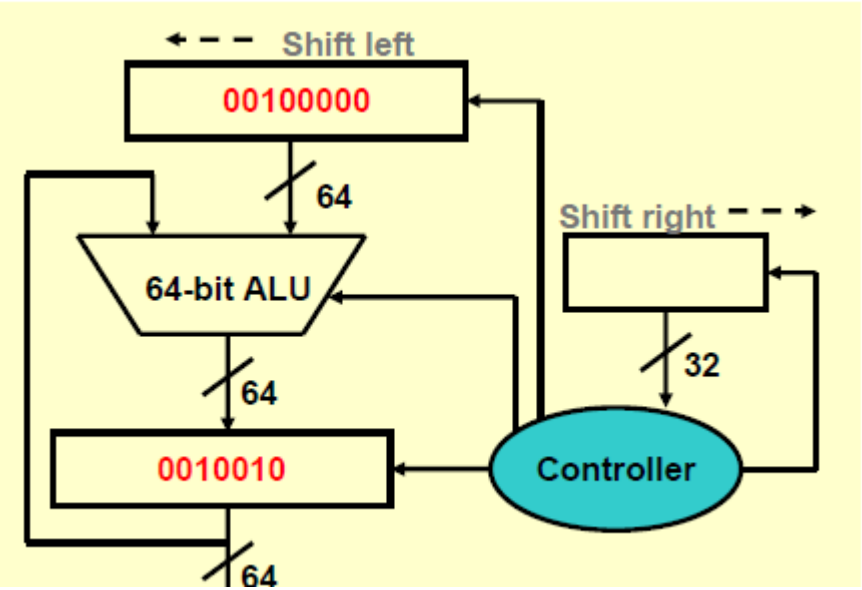
```
    0010   (multiplicand)
  x 1001   (multiplier)
  ------
    0010
   0000
  0000
+ 0010
  -------
  0010010  (product)
```

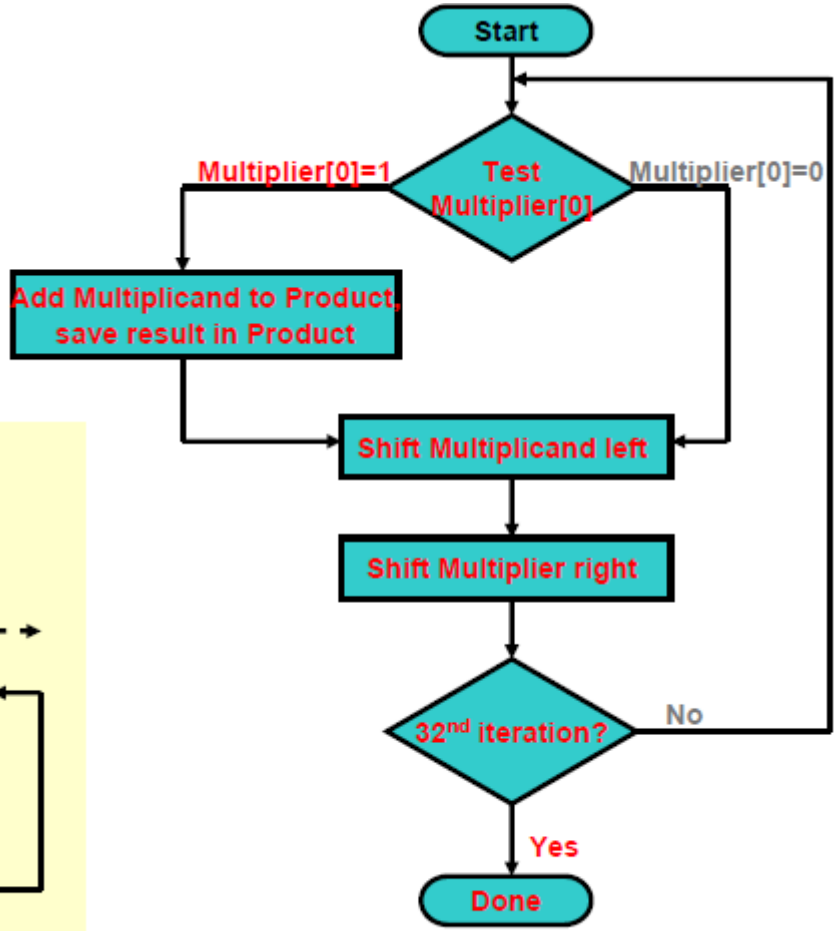




Control Flow

```
    0010   (multiplicand)
  x 1001   (multiplier)
  ------
    0010
   0000
  0000
+ 0010000
  -------
  0010010   (product)
```





Control Flow

Fix-point çarpma işlemi

Multiplicand 0010

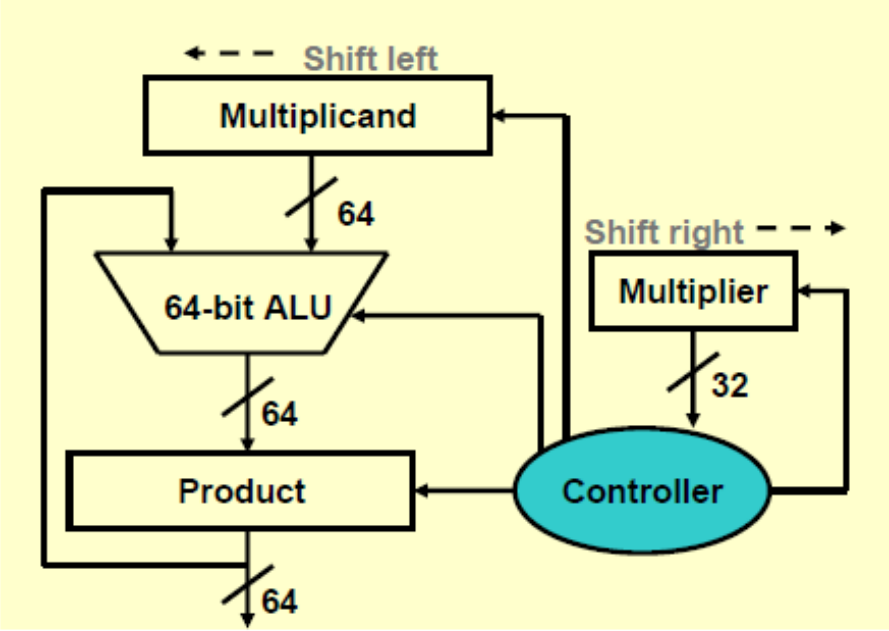
Multiplier x 0011

```
      0010
     0010
    0000
   0000
```

Product 00000110

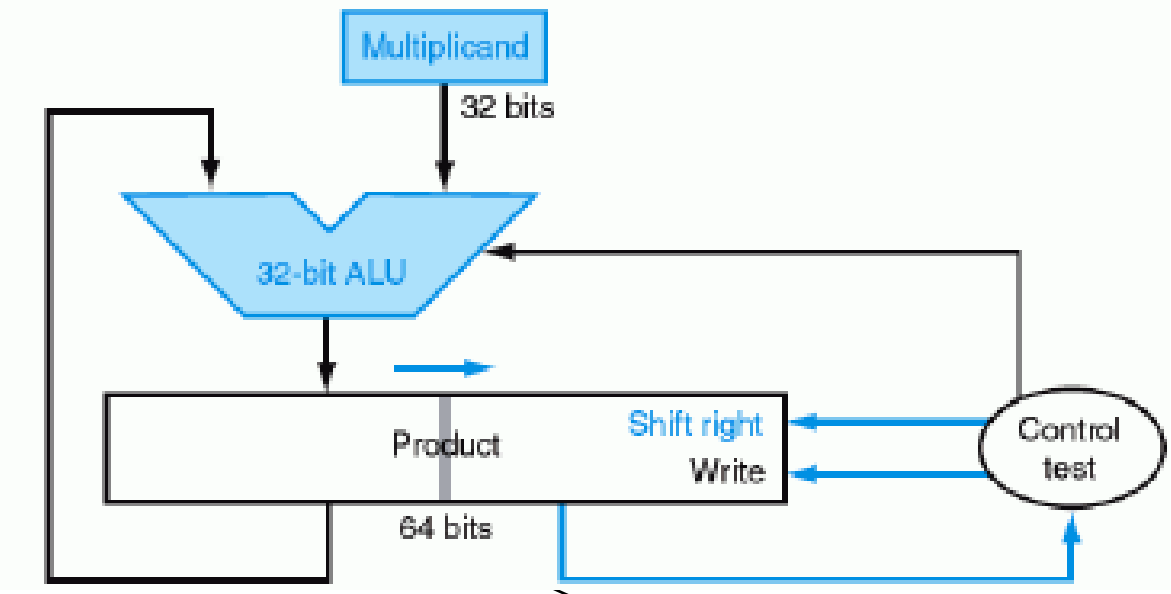
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Bu çarpma devresi işaretli sayıların Çarpılması için kullanılabilir mi?



İşaretli sayıların çarpılması işlemi; işaretsiz sayıların çarpılması şeklindedir. Yani negatif sayı pzitifsayıya dönüştürülüp iki işaretsiz ssayının çarpılması şeklinde gerçekjleştirilir. Sonucun işareti ise işaretli sayılarınişaret bitlerine göre belirlenir.

ÇARPMA İŞLEMİNİN İŞLENMİŞ VERSİYONU



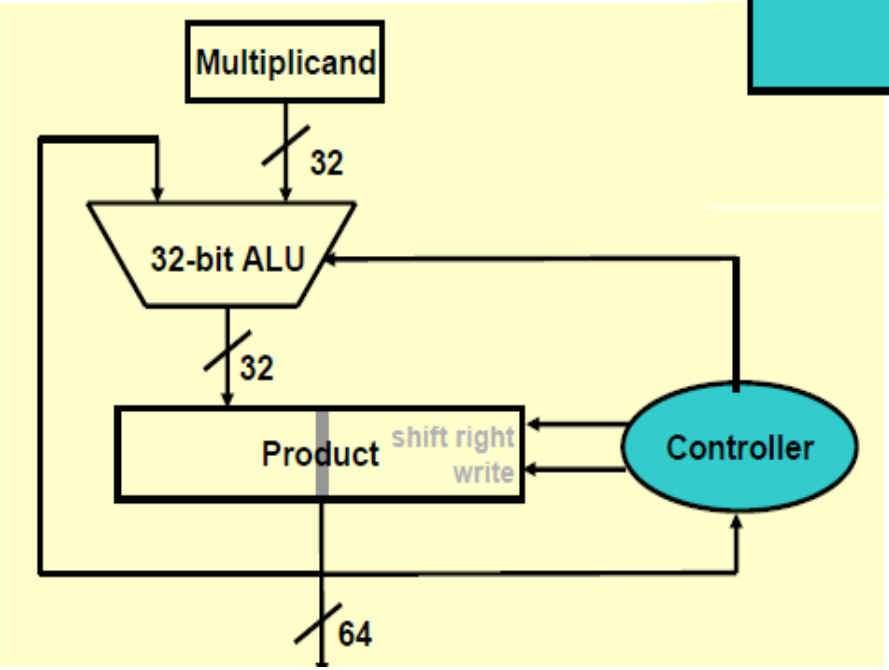
Çarpım register'ının başlangıç değeri 0

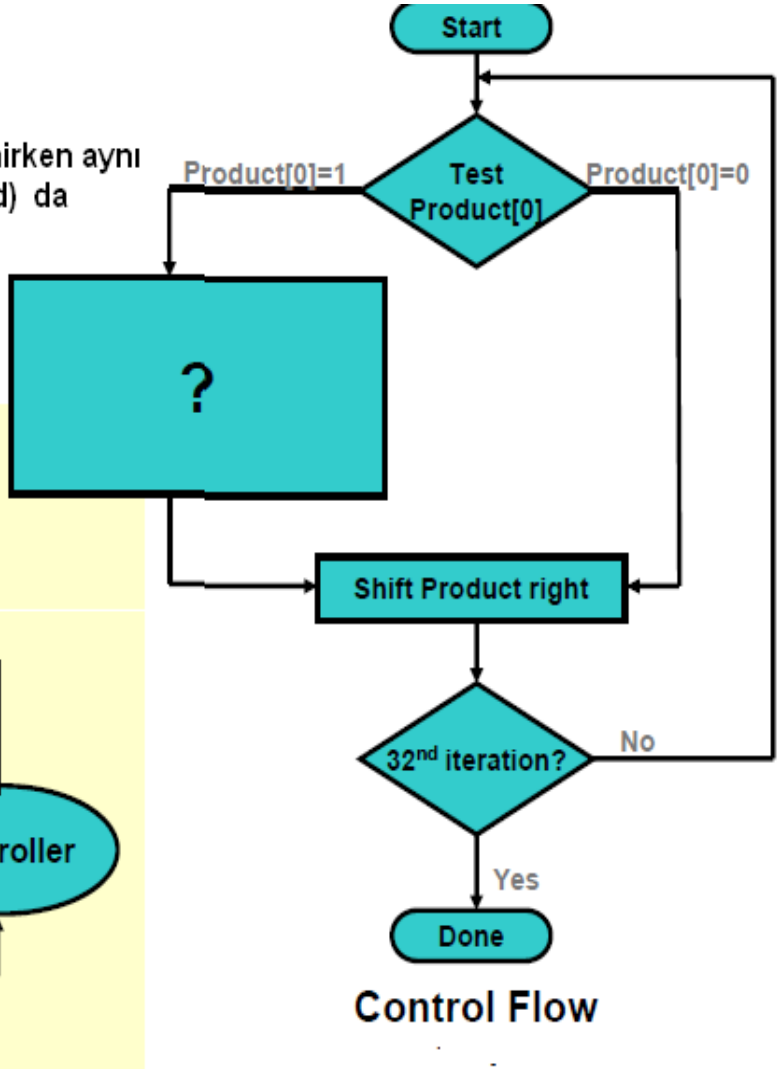
Multiplicand register, multiplier register ve ALU 32-bit uzunluğunda; product register ise 64-bit uzunluğundadır. Çarpım sonucu sağa kaydırılarak işlem devam eder.Çarpan bulunmadığı için multiplicand (çarpılan) çarpım register'ının en anlamlı 32 biti ile toplanır.

** Her adım(step) 1Cp'ına ihtiyaç duyar.Bu durumda 1.Çarpma devresine göre performans artmıştır.

**İşlem paralel başarılır.Çarpılan, Product reg'e eklenirken aynı zamanda Çarpan (multipler) ve çarpılan (multiplicand) da kaydırılır.

** Çarpıcı, Product reg'in ağırlıksız yarım 32 bitine yerleştirilmiştir.





Control Flow

İşlenmiş versiyona göre çarpma işlemi

Multiplicand 0010

Multiplier x 0011

```
  0010
 0010
0000
0000
```

Product 00000110

İterasyon	Step	Multiplier	Multiplicand	Product
0	init values	0011	0010	0000 0000
1	1a	0011	0010	0010 0000
	2	0011	0010	0001 0000
	3	0001	0010	0001 0000
2	1a	0001	0010	0011 0000
	2	0001	0010	0001 1000
	3	0000	0010	0001 1000
3	1a	0000	0010	0001 1000
	2	0000	0010	0000 1100
	3	0000	0010	0000 1100
4	1a	0000	0010	0000 1100
	2	0000	0010	0000 0110
	3	0000	0010	0000 0110

MIPS

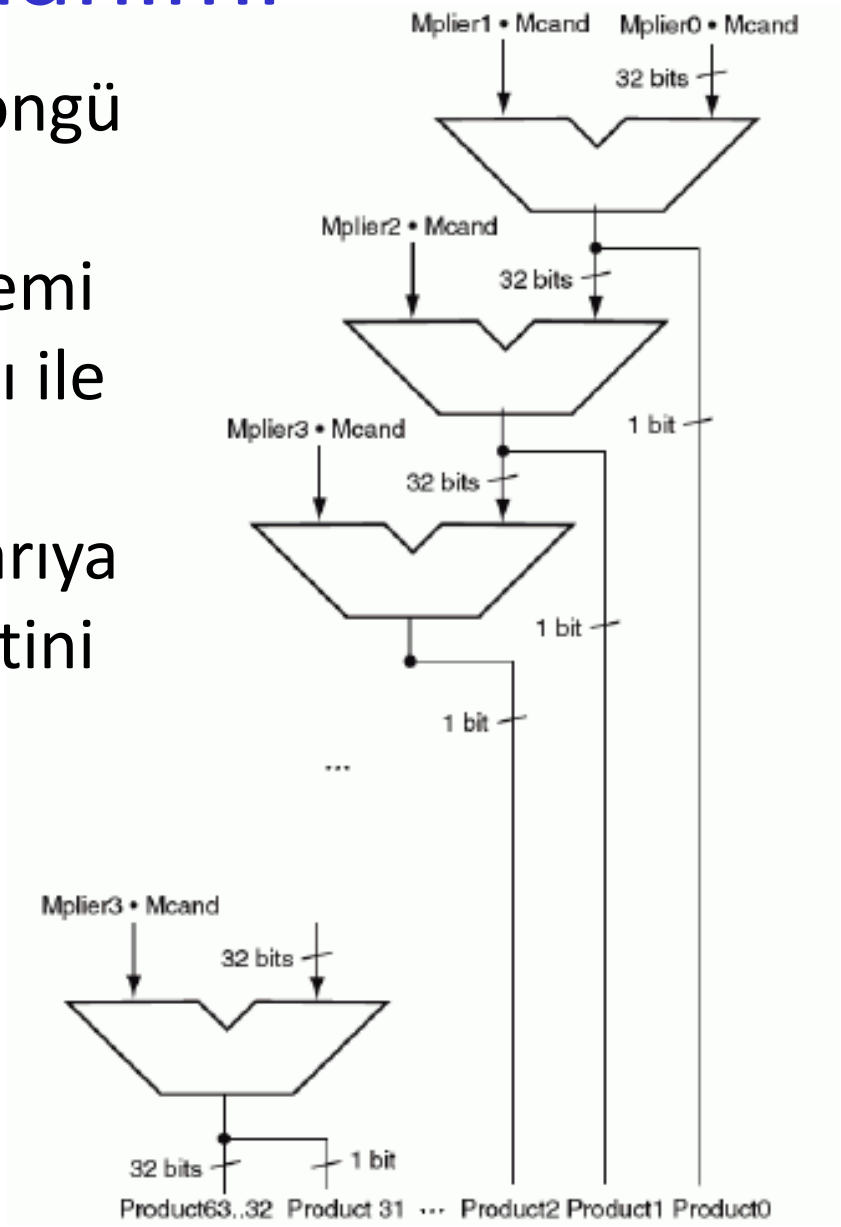
- **64 bitlik çarpımı ifade edebilmek için `Hi` ve `Lo` adı verilen 2 adet 32-bitlik register kullanılır.**
- **`mult, multu`**

  İşaretli ve işaretsiz sayılarda çarpma işlemi
- **`mflo, mfhi`**

  bu komutlar sanal (pseudoinstruction)komutlar olup yukarıda verilen mult ve multu komutları çalıştırıldığında bu komutlar 64 bitlik sonucu 32 bitlik Hi ve Lo olmak üzere 2 register'a taşır .

Hızlı çarpma donanımı

32 bitlik bir toplayıcı ile döngü kurarak 32 adet toplama yapmak yerine çarpma işlemi döngü açılarak 32 toplayıcı ile yandaki şekilde yapılabilir. Buradaki her toplayıcı dışarıya 32 bitlik toplam ile elde bitini göndermektedir.



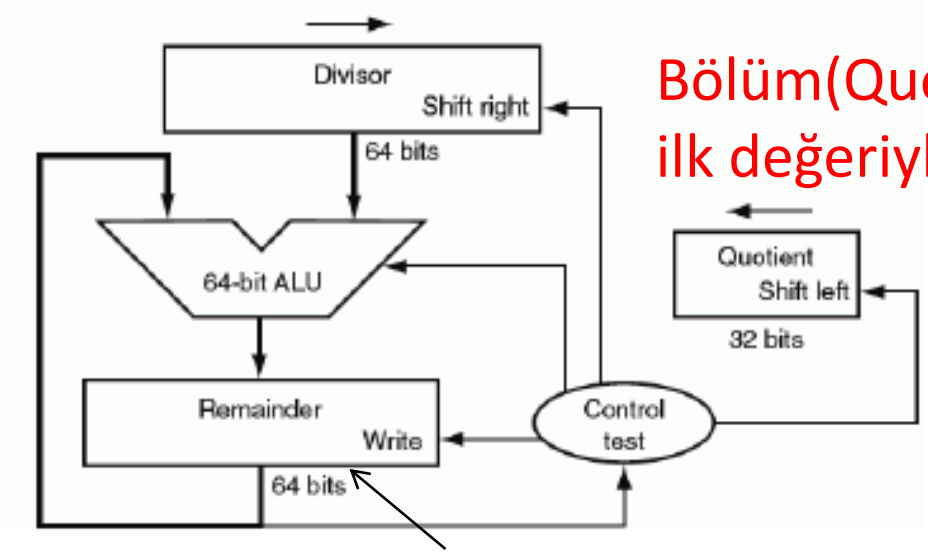
Bölme

```
                    1001    Quotient(Bölüm)
                 ┌──────
Divisor(Bölen) 1000 │1001010    Dividend (Bölünen)
                 -1000
                 ──────
                     10
                     101
                     1010
                    -1000
                    ─────
                       10    Remainder(Kalan)
```

Dividend = (Quotient * Divisor) + Remainder

Bölme donanımının ilk versiyonu

32 bitlik bölen (Divisor), bölen register'ının sol yarısından başlar ve her adımda sağa kaydırılır.

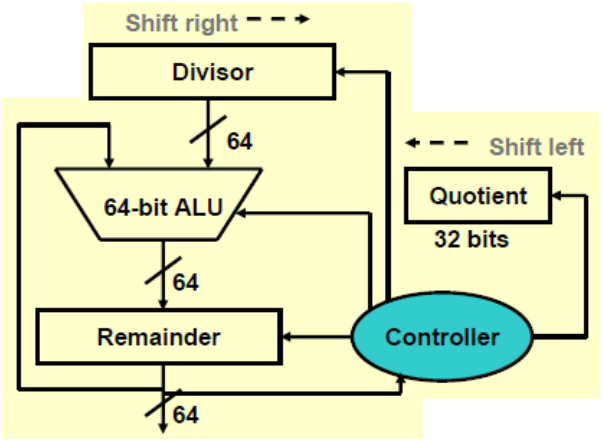


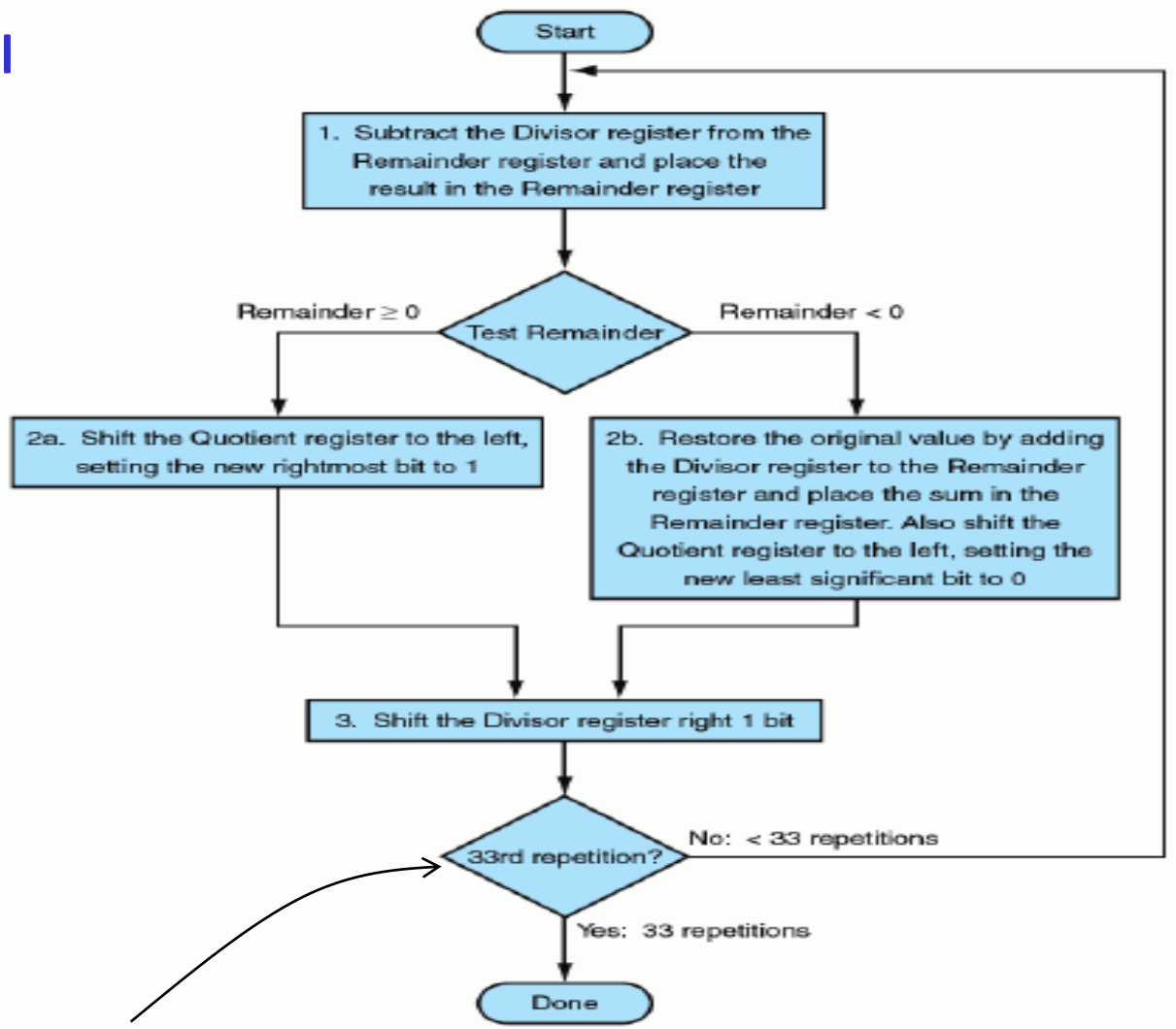
Bölüm(Quotient) registerı 0 ilk değeriyle başlamaktadır.

Kalan register'ı bölünen registerının değeriyle başlar.

Divisor(bölen) register, remainder(kalan) register ve ALU 64-bit uzunluğunda; quotient(bölüm) registerı 32-bit uzunluğundadır. Kontrol bölen ve bölümün ne zaman kaydırılacağı ve kalan registerına yeni değerin ne zaman yazılacağını belirler.

Bölme Algoritması



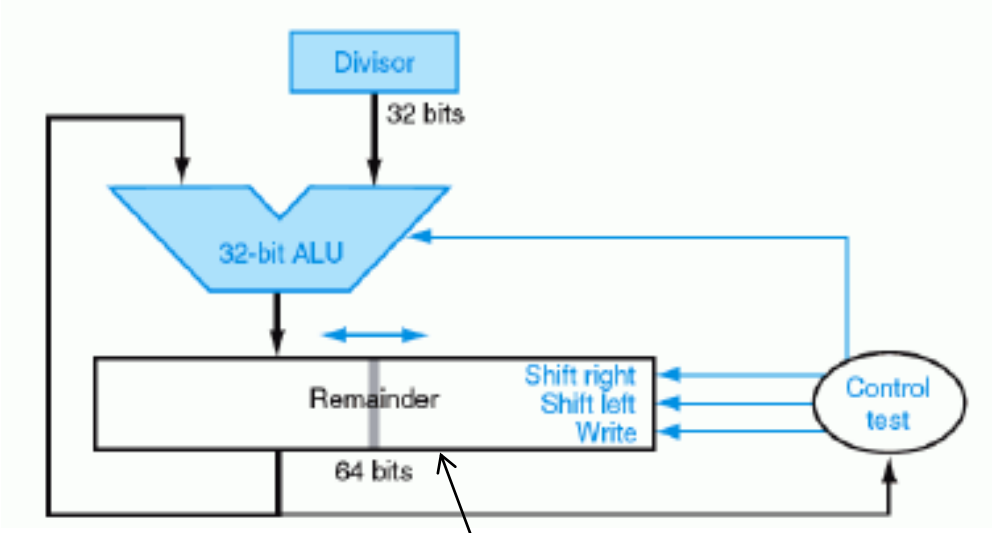


Neden 33?

00000111_2 (7)/ 0010_2 (2)

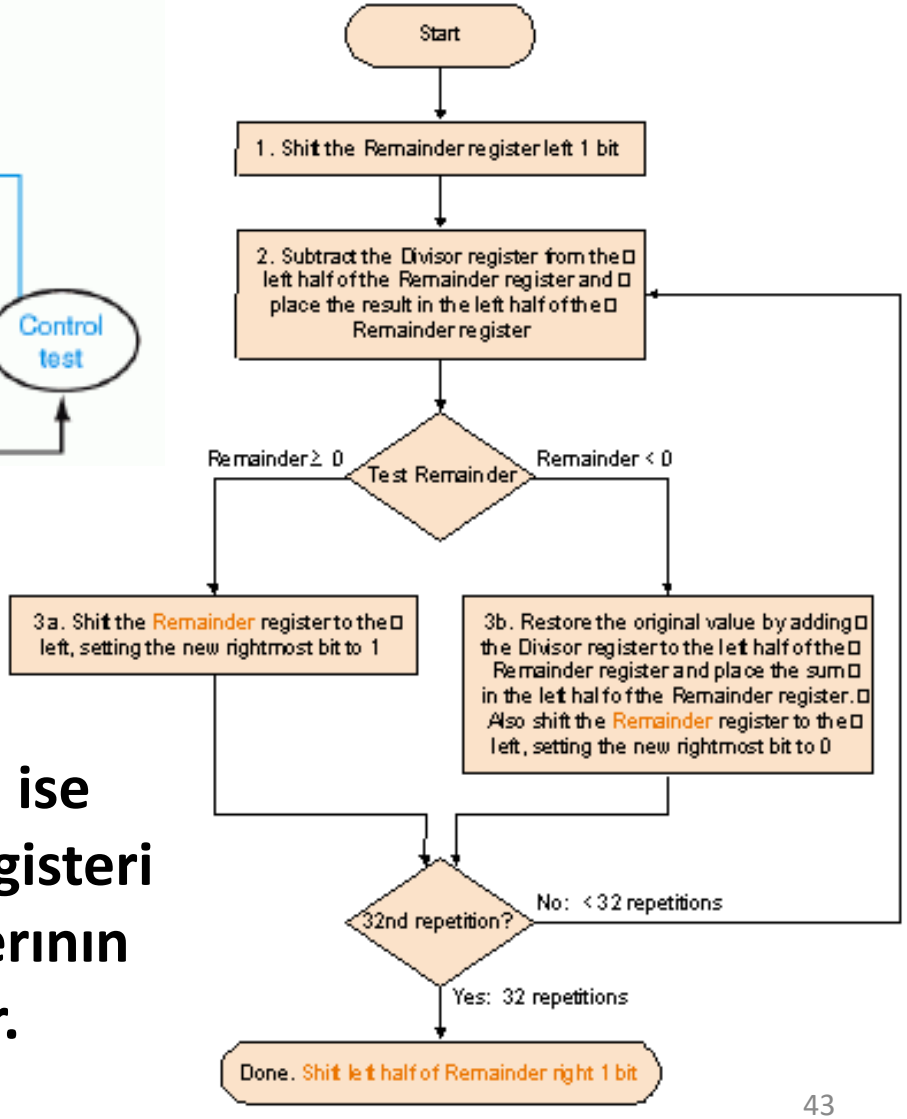
Iteration	Step	Bölüm Quotient	Bölen Divisor	Kalan Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Bölme donanımının gelişmiş versiyonu



Kalan registerının anlamlı bitlerinde bölüm registerının değeri bulunur.

Bölen, ALU ve bölüm 32 bitlik, kalan ise 64 bitliktir. Bu donanımda bölüm registeri bulunmamakta, bölüm kalan registerının en anlamlı bitlerinde bulunmaktadır.



MIPS

- **div, divu**

  iki tane 32 bitlik operand registerı ile işlenen içerik bölünür, kalan Hi registerına bölüm ise Lo registerına aktarılır taşma her iki durumda da göz ardı edilir.

  Sözde talimatlar kullanılarak div (signed with overflow), divu (unsigned without overflow) 3 tane 32 bitlik register ile iki registerın bölümü 3 . registera aktarılır.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	lhu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh $s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte unsigned	lbu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FLOATING POINT SAYILAR

(KAYAN NOKTALI SAYILAR)

Floating Point Sayılar

- Gerçek sayılar tamsayı ve kesirli kısımdan oluşur. Kesir kısmı sayının hassasiyetini, tamsayı kısmı büyüklüğünü ifade eder (91023,00002)
- Bilgisayarda (Binary formatta)g erçek sayıları temsil etmek oldukça önemlidir. (Gerçel sayıların tamsayı ve kesir kısımları hafızada nasıl tutulacaktır.)

1- Fixed point (Sabit noktalı) Sayı gösterimi : Virgüllü sayılar için sayının tam sayı ve kesirli kısımları için önceden belirlenmiş uzunlukta bit sayısı ayrılır. Sabit noktalı sayılar hafızada fazla yer kaplar, işlemleri uzundur, yeteri kadar hassas olamayabilirler.

$138{,}005_{10}$ sayısının 8.8-bit Fixpoint gösterimi = 10001010.00000001_2

= 138,0039062'dır (8 bitlik kesir kısmı ancak bu kadar yaklaşabiliyor.)

2- Floating point (Kayan noktalı) Sayı gösterimi: Gerçel sayıların bilimsel formatta yani üstel (exponantial) sayı formatında gösteriminin binary sözcük şeklinde ifade edilmesidir. Buna IEEE754 Floating point formatı denir. IEEE 754 floaing point formatı günümüzde gerçek sayıları temsil etmek üzere kullanılan en yaygın gösterim şeklidir. Bu gösterim şekli çok büyük sayı aralığında neredeyse sabit kesinlik oranı sağlar.

$138{,}005 = 1.38005 \times 10^2$ (Üstel gösterim – Bilimsel Gösterim -Bu formatta Taban'ın kuvveti değiştikçe çarpan sayının tam sayı kısmını belirleyen nokta, kuvvetin değişme yönüne uygun olarak sağa veya sola kayar.)

= 430A0148 = 0 10000110 000 1010 0000 0001 0100 1000 (IEEE754 FP ifade)

$= (0)^0 \times 2^7 \times 1.00010100000000101001000 = 138.005$

Floating Point

Aşağıda verilen sayıları tanımlayabilmek için floating point sayılar kullanılır;

- Kesirli sayılar, örneğin 3.1416
- Çok küçük sayılar, örneğin .0000000000023
- Ve çok büyük sayılar, örneğin $-3.15576 * 10^{46}$

Floating point sayı gösterimi

Floating point sayıların gösterimi 3 bölümden oluşur.

Sign Biti (İşaret Biti): Sign biti pozitif sayılar için 0, negatif sayılar için 1 değerini alır.

Exponent (Üs): Exponent alanı hem negatif hem de pozitif üsleri temsil edebilmektedir.Bunu gerçekleştirmek için bir **bias değeri** gerçek üs değeriyle toplanıp exponent kısmı oluşturulur.

IEEE 32 bitlik gösterimi için ***bias değeri*** 127, 64 bitlik gösterim için de 1023 tür.Buna göre gerçek üssün 0 olması exponent alanında saklanan değerin 127 olacağı anlamına gelir.

Mantissa: (Fraction – Kesir) Mantissa sayıyı ifade eden bitleri gösterir. Bu,sayının tam ve kesir (fraction- mantisa) kısımlarını gösteren bitlerden oluşur.

Floating point sayılar aşağıdaki gibi gösterilir:

$$(-1)^{sign} * significand * 2^{exponent}$$

Örneğin; -101.001101 * 2^{111001} (Normalize edilmemiş)

- *significand* bölümü ne kadar bit ile ifade edilirse sayı gerçeğe o kadar yakın olur.
- *exponent* bölümü ne kadar bitle ifade edilirse, değer aralığı o kadar artar.

Floating point sayılar aşağıdaki gibi gösterilir:

$$(-1)^{sign} * significand * 2^{exponent}$$

Örneğin; $-101.001101 * 2^{111001}$

- *significand* bölümü ne kadar bit ile ifade edilirse sayı gerçeğe o kadar yakın olur.
- *exponent* bölümü ne kadar bitle ifade edilirse, değer aralığı o kadar artar.

Floating point sayılar $\mathbf{1 \leq significand < 10_{two}}(=2_{ten})$ aralığında ifade edilir. Bu aralıkta olmayan sayıların normalize edilmeleri gerekmektedir.Örneğin; $\mathbf{-101.001101 * 2^{111001}}$ sayısının floating point olarak gösterilişi (normalize edilmiş hali),

$\mathbf{-1.01001101 * 2^{111011}}$ şeklindedir.

Floating point standartında gösterilecek olan sayıların normalize edilmesi zorunludur. Sayıların normalize edilmesindeki amaç aynı sayıyı daha az bit ile ifade etmektir. Bu nedenle binary formda ifade edilecek her sayı aşağıdaki formatta olmak zorundadır.

$$1.xxxxxxx_2 \times 2^{yyyyy}$$

Örneğin binary bir sayı olan

11110000 11001100 10101010 00000000;

+ 1.1110000 11001100 10101010 x 2^{31}

şeklinde gösterilir.

NORMALİZASYONUN (OLAĞANLAŞTIRMANINI) AVANTAJLARI

- Olağanlaştırma ile her sayı için benzersiz bir gösterim sağlanır.
- Virgülün solunda kalan sayı 1 olduğu için sayının anlamlı kısmında bir bit'lik tasarruf sağlanmış olur.

17.1875

10001.0011×2^0 Saf ikili sayı formatı

1.00010011×2^4 Olağanlaştırılmış ikili sayı formatı

0	10100	00010011

IEEE 754 Floating-point Standard

IEEE 754 floating point standartında 32 bitlik ve 64 bitlik olmak üzere 2 tür gösterim mevcuttur.Her 2 gösterim için sign, exponent ve mantissanın durumları aşağıda verilmiştir.

- Single precision (tek duyarlıklı): tek kelime (32 bit)

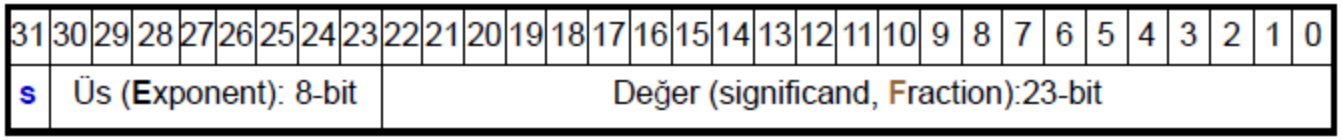
31	bits 30 to 23	bits 22 to 0
sign	8-bit exponent	23-bit fraction

- Double precision (çift duyarlıklı): iki kelime (64 bit)

31	bits 30 to 20	bits 19 to 0
sign	11-bit exponent	upper 20 bits of 52-bit fraction

bits 31 to 0
lower 32 bits of 52-bit fraction

Tek Duyarlı Kayan Nokta (Single)

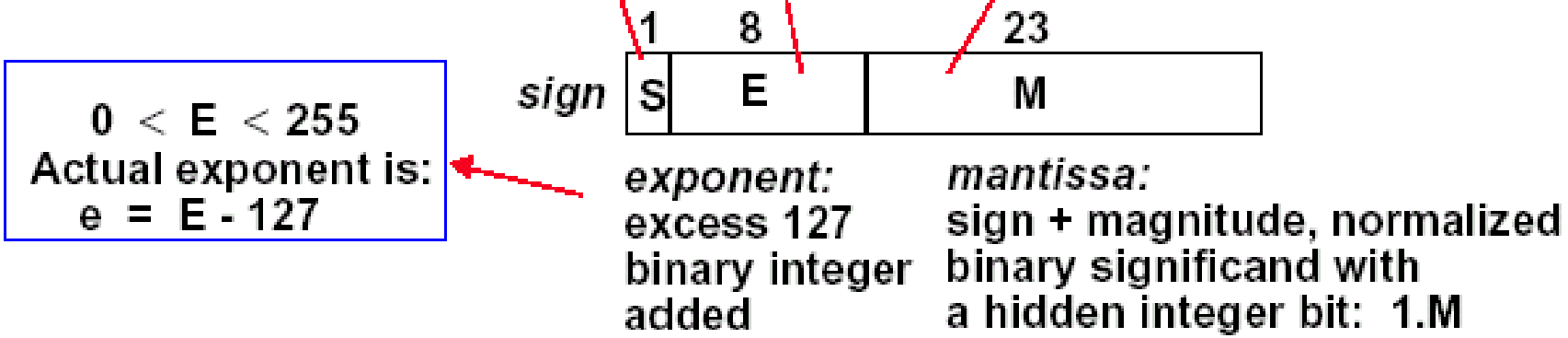


$FP = (-1)^s \times (1+F) \times 2^{E\text{-bias}}$ {Sıfır Sayısı istisnaidir. Bütün bitler 0}

- Üst saptırma miktarı (bias) 127 olarak belirlenmiştir. (Üs=E-bias)
  - Bias (saptırma) aralığı –126 ila 127 arasındadır.
  - 127'den büyük sayılar pozitif üstlerdir.
  - 127'den küçük sayılar negatif üstlerdir.
  - -126 sayısı 1 olarak gösterilir (0000 0001)
  - 0 sayısı 127 olarak gösterilir (0111 1111)
- 0 değeri hariç FP formata donanımsal "1" değeri eklenir,böylece significand kısım:Single için: 23+1 =24-bit, Double için: 52+1=53-bit
- Tek duyarlı kayan nokta sayılar $\{2.0 \times 10^{-38} \ldots\ldots 2.0 \times 10^{38}\}$

Representation of Floating Point Numbers in Single Precision *IEEE 754 Standard*

$$\text{Value} = N = (-1)^S \times 2^{E-127} \times (1.M)$$



Example: 0 = 0 00000000 0 . . . 0 -1.5 = 1 01111111 10 . . . 0

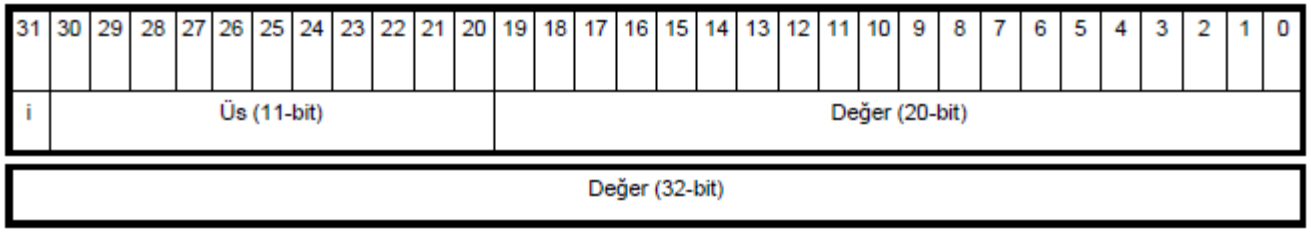
Magnitude of numbers that can be represented is in the range:

$$2^{-126}(1.0) \quad \text{to} \quad 2^{127}(2 - 2^{-23})$$

Which is approximately:

$$1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}$$

Çift Duyarlı Kayan Nokta (Double)



$FP = (-1)^s \times (1+F) \times 2^{E\text{-bias}}$ {Sıfır Sayısı istisnaidir. Bütün bitler 0}

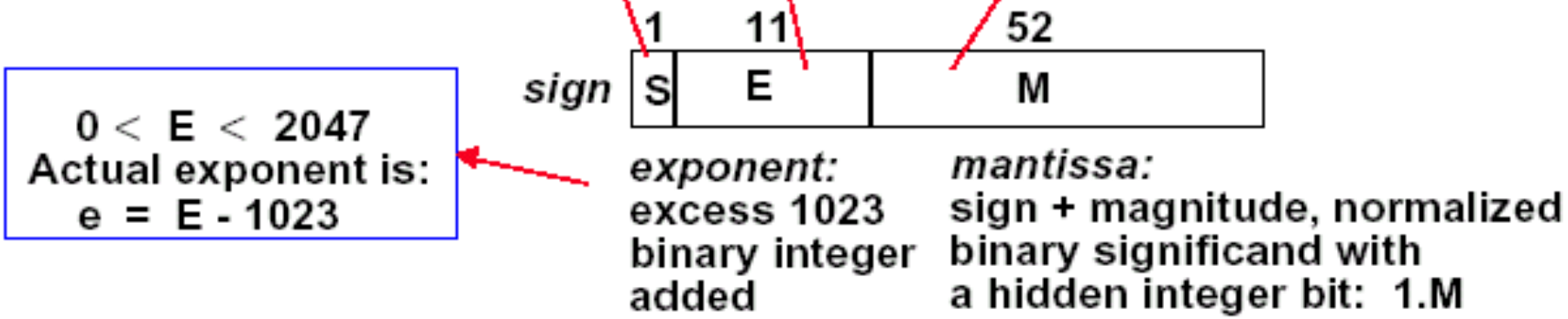
- Çift duyarlı gösterimde ise 64 bitin 1 biti işaret, 11 biti üst ve 53 biti de anlamlı kısmı ifade etmek için kullanılır.

- Üst saptırma değeri 1023 olarak belirlenmiştir.
  - 1023'den büyük olan sayılar pozitif, küçük olan sayılar ise negatif üstleri göstermektedir.
  - Üs=E-bias

- Çift duyarlı: 2.0×10^{-308}.....2.0×10^{308}

Representation of Floating Point Numbers in Double Precision *IEEE 754 Standard*

$$\text{Value} = N = (-1)^S \times 2^{E-1023} \times (1.M)$$



Example: 0 = 0 00000000000 0 . . . 0 -1.5 = 1 01111111111 10 . . . 0

Magnitude of numbers that can be represented is in the range: $2^{-1022}(1.0)$ to $2^{1023}(2 - 2^{-52})$

Which is approximately: 2.23×10^{-308} to 1.8×10^{308}

IEEE 754 Floating-point Standard

- Sayı normalize kabul edilir ve sayıda noktanın sol tarafında yer alan 1 biti fraction kısmında gösterilmez.
  - Yani significand kısmı 1.1001… olan sayı 1001… şeklinde ifade edilir.
  - Sayıların temsili:

$$\text{sayı değeri} = (-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{\text{exponent value}}$$

- Sayının değerini daha kolay bir şekilde bulabilmek için exponent biaslanır.
  - Exponent tamamen 0lardan oluşmuşsa en küçük, 1'lerden oluşmuşsa en büyüktür.
  - single precision için bias değeri 127, double precision ise 1023'tür.
  - Sayı temsili;

$$\text{sayı değeri} = (-1)^{\text{sign}} * (1 + \text{fraction}) * 2^{(\text{exponent} - \text{bias})}$$

Örnek 1;

-0.75_{ten} sayısını IEEE 754 standartına göre single precision ve double precision formatında ifade edelim.

Desimal değer: $-0.75 = -3/4 = -3/2^2$

Binary karşılığı: $-11/100 = -.11 = -1.1 \times 2^{-1}$

IEEE single precision floating point exponent

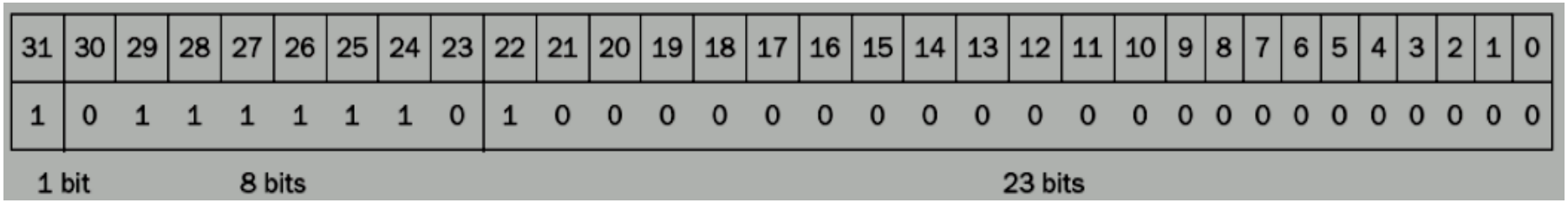
= bias + exponent değeri

$= 127 + (-1) = 126_{ten}$

$= 01111110_{two}$

IEEE 754 single precision formunda 0.75:

$(-1)^{1}*(1 +.1000\ 0000\ 0000\ 0000\ 0000\ 0000) * 2^{(126-127)}$



IEEE 754 double precision formunda 0.75:

$(-1)^{1} *(1 +.1000\ 0000\ 0000\ 0000\ 0000…0000)*2^{(1022-1023)}$

Örnek 2; -2345.125_{10} desimal sayısını IEEE 754 standartında 32 bitlik formatta ifade edelim:

$-2345.125_{10} = -100100101001.001_2$ (binary karşılığı)

$= -1.00100101001001_2 \times 2^{11}$ (normalize binary)

yazılmaz

-Mantissa negatif olduğu için işaret biti 1 olacaktır. (S=1)

-32 bit için bias değeri 127 idi, binary sayının exponenti=11

$E=127+11=138_{10} = 10001010_2$ (8 bit)

-Kesir kısmı (fraction)= .00100101001001000000000 (23 bit)

Sign	Exponent	Fraction
1	10001010	00100101001001000000000
1bit	8bit	23bit

Örnek 3; Aşağıda verilen ikilik tabandaki floating point sayıyı desimal sayıya çevirelim.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

İşaret biti:1, exponent alanının değeri:129,

fraction alanı: $1x2^{-2} = ¼ = 0.25$

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} = (-1)^1 \times (1 + 0.25) \times 2^{(129-127)}$$
$$= -1 \times 1.25 \times 2^2$$
$$= -1.25 \times 4$$
$$= -5.0$$

IEEE 754 formatında sayı gösterimi

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1–254	anything	1–2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

0 özel durumu:

- Exponent ve fraction tamamen 0'lardan oluşmuşsa, işaret biti ne olursa olsun sayının değeri 0'dır.
- Exponent 0 ve fraction 0'dan farklı ise sayının değeri;

$$(-1)^{sign} * (1 + fraction) * 2^{-127}$$ 'dir.

Matlab ortamında floating-point sayılar

```
>> q= quantizer('single','nearest','saturate');
```

Floating-point sayının özelliklerinin belirlendiği fonksiyon

```
>> num2hex(single(-0.75))
ans =  bf400000
```

Desimal sayının floating point karşılığını veren komut

1 01111110 10000000000000000000000

floating point sayının desimal karşılığını veren komut

```
>> hex2num(q,'C5129200')
```

1 10001010 00100101001001000000000

```
ans = -2.3451e+003
```

Floating Point Toplama

Örnek; $1{,}234{,}823.333_{10} + .0011_{10} = ?$

Floating point sayıların toplanabilmesi için öncelikle kesir kısımlarının toplanması gerekir ancak toplamaya başlanılmadan önce sayıların exponentlerinin aynı olması sağlanmalıdır. Bu nedenle toplama işlemine geçilmeden önce üslerin eşitlenmesi gerekir.

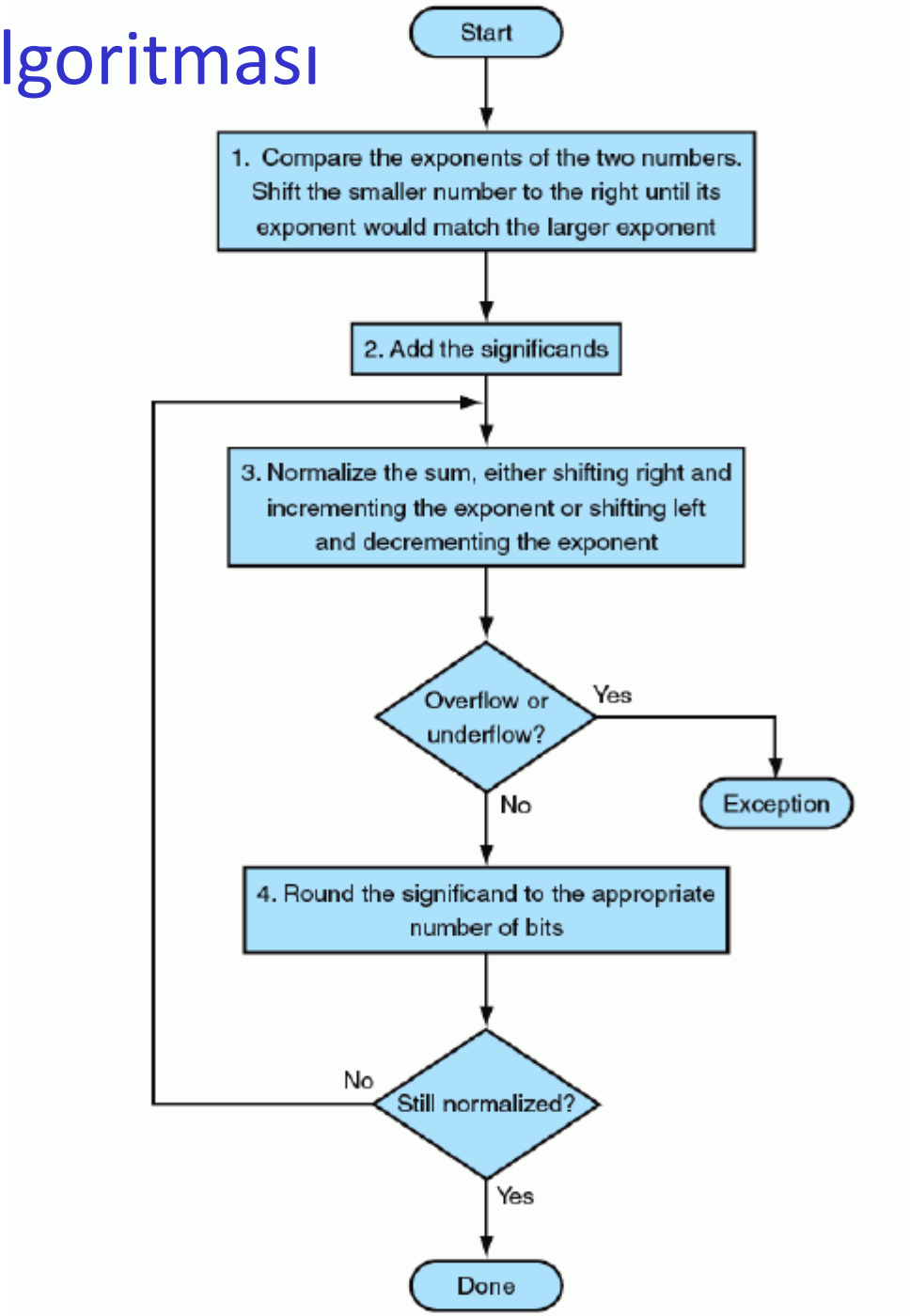
$1{,}234{,}823.333 = 1.234823333 \times 10^{6}$

$.0011 = 1.1 \times 10^{-3} = 0.0000000011 \times 10^{6}$

Toplanacak sayıların üsleri eşitlendikten sonra önce kesir kısmı ardından tam kısımları toplanır.

$$
\begin{array}{rl}
 & 1.2348233330 \times 10^{6} \\
+ & 0.0000000011 \times 10^{6} \\
\hline
 & 1.2348233341 \times 10^{6}
\end{array}
$$

Toplama işleminin Algoritması



Verilen algoritmaya göre toplama işleminin adımları;

- **1. adım:** toplanacak olan sayıların exponentleri eşitlenir. küçük olan sağa kaydırılarak büyük olan exponente eşitlenir.

- **2.adım:** sayıların fraction(mantisa) kısımları toplanır.

- **3. adım:** elde edilen toplam normalize edilir.

- **4. adım:** elde edilen toplam değeri olması istenen uzunluğa(4 bit) getirilir.

Örnek 2; 0.5 + (-0.4375) =?

Toplam 4 dijitle ifade edilsin.

Öncelikle sayıların binary karşılıklarını hesaplayalım:

$$
\begin{aligned}
0.5_{ten} &= 1/2_{ten} &&= 1/2^{1}{}_{ten} \\
&= 0.1_{two} &&= 0.1_{two} \times 2^{0} && = 1.000_{two} \times 2^{-1} \\
-0.4375_{ten} &= -7/16_{ten} &&= -7/2^{4}{}_{ten} \\
&= -0.0111_{two} &&= -0.0111_{two} \times 2^{0} && = -1.110_{two} \times 2^{-2}
\end{aligned}
$$

1. adım: exponentlerin eşitlenmesi

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

2. adım: fraction kısımlarının toplanması

$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$

3. adım: toplamın normalize edilmesi

$$\begin{aligned} 0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\ &= 1.000_{\text{two}} \times 2^{-4} \end{aligned}$$

4. adım: overflow,underflow oluşmuş mu?

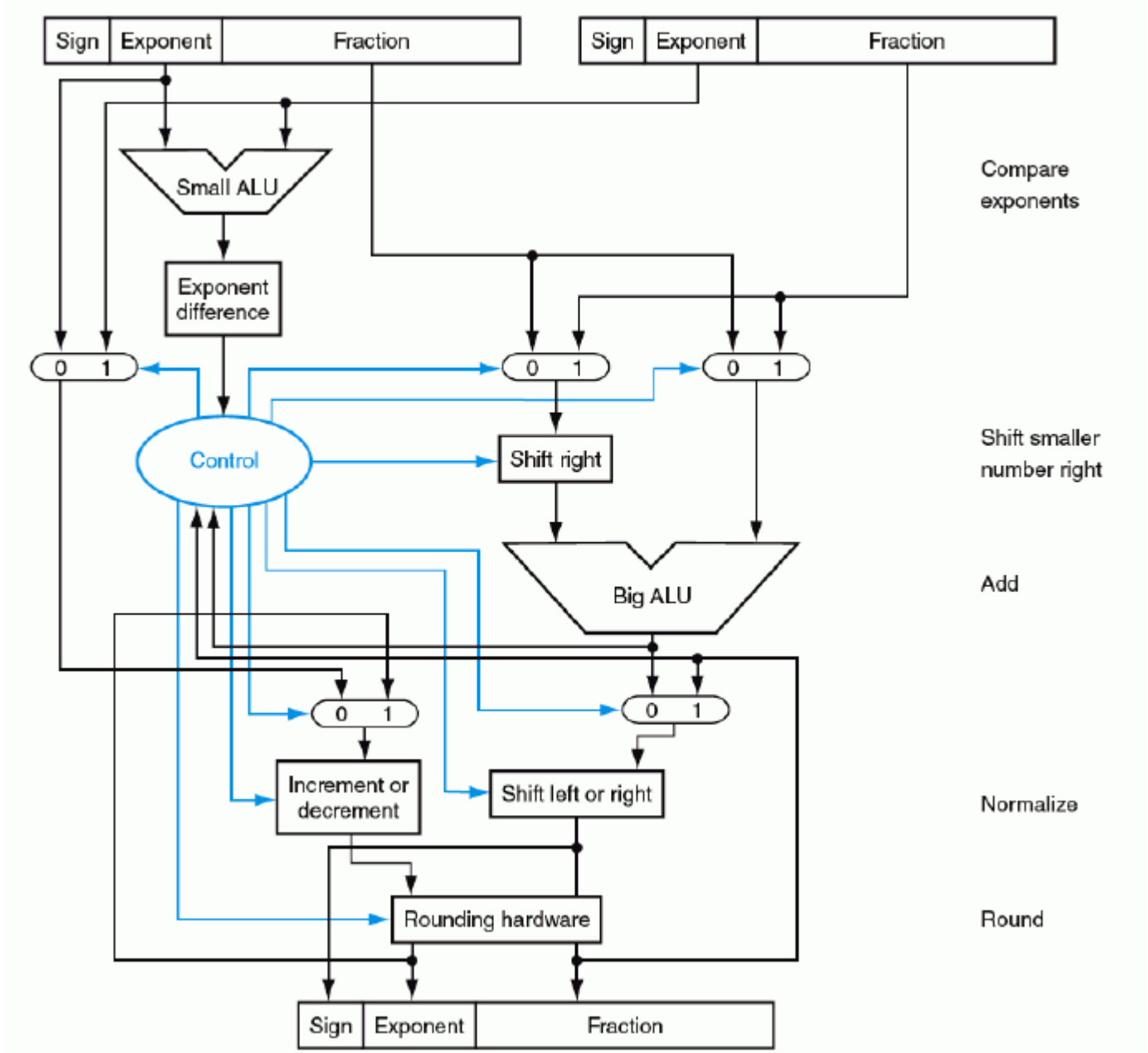
Üs değeri 127≥-4≥-126 arasında olduğu sürece overflow veya underflow oluşmaz.

$$1.000_{two} \times 2^{-4}$$

Sonuç olarak elde edilen değerin desimal karşılığı;

$$\begin{aligned} 1.000_{two} \times 2^{-4} &= 0.0001000_{two} = 0.0001_{two} \\ &= 1/2^{4}{}_{ten} = 1/16_{ten} = 0.0625_{ten} \end{aligned}$$

Floating point toplama için donanımın blok şeması



Örnek; 2345.125_{10} sayısı ile $.75_{10}$ sayısını IEEE 754 standartında 32 bitlik formatta toplayınız.

$X = 2345.125_{10}$

Sign	Exponent	Fraction
0	10001010	00100101001001000000000

$Y = 0.75_{10}$

Sign	Exponent	Fraction
0	01111110	10000000000000000000000

1.adım: Exponentlerin büyük olana eşitlenmesi

$E_x > E_y$ $E = 10001010 = 138_{10}$

$E_x - E_y = 10001010 - 01111110 = 00000110 = 12_{10}$

Fy 12 bit sağa kaydırılacak.

$F_y \times 2^{-12} = 0.000000000011000000000$

2.adım: kesirlerin toplanması

$Fx + (Fy \times 2^{-12}) = 1.\ 00100101001001000000000$

$+ 0.\ 00000000001100000000000$

$= 1.00100101001111000000000$

3. adım: toplam normalize mi? Evet

4.adım: overflow veya underflow oluşmuş mu?hayır

5.adım: sonuç 0 mı? Hayır

Sonuç:

Sign	Exponent	Fraction
0	10001010	00100101001111000000000
1bit	8bit	23bit

Floating point sayılarda Çarpma

Örnek 1; $1.110_{10} x 10^{10} x 9.200_{10} x 10^{-5}$ dört dijitlik sayıları çarpalım (IEEE 754 single precision).

1.adım: toplamadan farklı olarak üsler toplanır. Öncelikle biaslanmış exponentleri hesaplamalıyız.

10+127=137 ve (-5)+127= 122

Yeni exponent= 137+122= 259

259 sayısını 8 bitle nasıl ifade edeceğiz?

Biaslanmış sayıları toplarken bias değerini çıkarmayı unutmamamız gerekir, dolayısıyla;

yeni exponent= 137+122-127 = 132

ya da yeni exponent= 10 + (-5)+ 127 = 132 olur.

2.adım: significand kısımları çarpılır.

$$
\begin{array}{r}
1.110 \\
\times\ 9.200 \\
\hline
0000 \\
0000 \\
2220 \\
+\ 9990 \\
\hline
10212000_{10}
\end{array}
$$

Çarpım sonucu= $10.212000_{10}=10.212_{10}x10^5$

3. adım: sonucun normalize edilmesi

$$10.212_{10} x 10^5 = 1.0212_{10} x 10^6$$

4.adım: significant kısmı 4 dijit olduğu için sayının yuvarlanması gerekir.

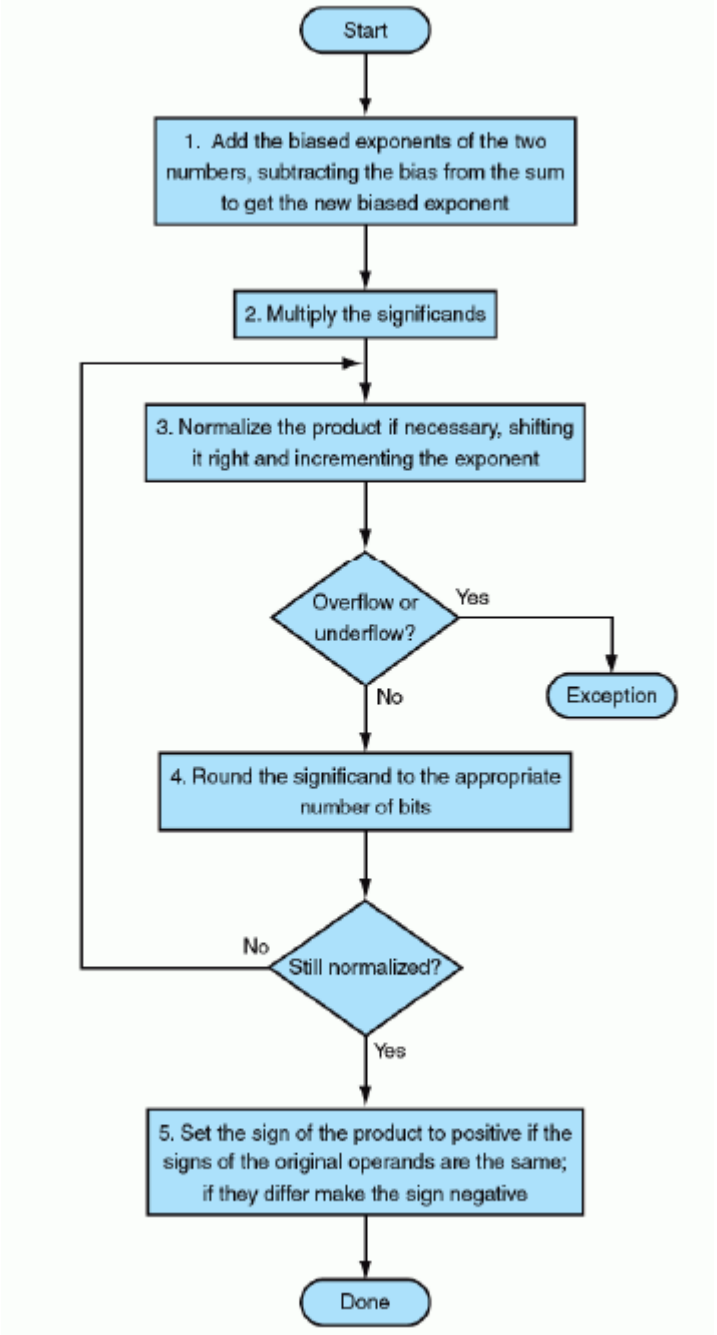
$$1.0212_{10} x 10^6 = 1.021_{10} x 10^6$$

5.adım: işaret bitinin değerinin bulunması. Çarpılan sayıların işaretleri aynı ise sonuç pozitif(S=0), farklı ise sonuç negatif (S=1) olur.

Sonuç= $+1.021_{10} x 10^6$

Çarpma Algoritması

Genel olarak 3 ve 4. adımlar birer kez gerçekleştirilir, ancak normalizasyon yapılmamışsa 3. adım tekrarlanmalıdır.



Örnek 2; Verilen algoritmanın adımlarını takip ederek 0.5_{10} ile -0.4375_{10} sayılarını çarpalım.

Sayıların binary karşılıkları; $1.000_2 x 2^{-1}$ ve $-1.110_2 x 2^{-2}$

1. Adım: exponentlerin toplanması

Bias olmadan -1 +(-2)= -3

Bias ile; (-1+127)+(-2+127)-127=124

2. Adım: significand kısımlarının çarpımı

$$
\begin{array}{r}
1.000_2 \\
\times\ 1.110_2 \\
\hline
0000 \\
1000 \\
1000 \\
+\ 1000 \\
\hline
1110000_2
\end{array}
$$

çarpım= $1.110000 x 2^{-3}$

3. Adım: çarpım sonucu normalize mi? **Evet**

İşlem sonucunda overflow oluşmuş mu?

127≥-3≥-126 aralığında olduğundan **hayır**

Bias edilen için 254≥124≥1 aralığında olduğundan **hayır**

Underflow oluşmuş mu? **Hayır**

4. Adım: çarpım değerini 4 bit ile ifade etmemiz gerekiyor. Dolayısı ile çarpım= $1.110000x2^{-3}=1.110x2^{-3}$

5. Adım: çarpılan sayıların işaretleri farklı olduğu için sonuç negatif olacaktır.

Sonuç = $-1.110x2^{-3}$

Sonucu onluk tabana çevirerek değerini kontrol edelim;

Sonuç = $-1.110x2^{-3}=-0.001110_2=-0.00111_2$

$=-7/2^5{}_{10}=-7/32_{10}=\mathbf{-0.21875_{10}}$

Örnek 3; -18_{10} ve 9.5_{10} sayılarını IEEE 754 single precision formatında çarpalım.

$X = -18 = 10010_2$

1	10000011	00100000000000000000000

$Y = 9.5 = 1001,1_2$

0	10000010	00110000000000000000000

(1) Exponentleri topla

$E_x + E_y - 127_{10} = 1000\ 0011 + 1000\ 0010 - 01111111$

$= 1000\ 0110$

(2) Fraction kısımlarının çarpımı

1.0010 0000… 0000 x 1.0011 0000 … 0000

= 01. $\underbrace{0101\ 0110\ 0000 \ldots 0000\ 0000}$

46 bit uzunluğunda

Sonuç single precision olacağı için fraction kısmı 24 bit uzunluğuna indirilmelidir.

Fraction = **1.0101 0110 0000 0000 0000 000**

(3) Fraction kısmının normalize edilmesi gerekmiyor.

(4) overflow? **Yok** underflow? **Yok**

(5) İşaret biti

Sx XOR Sy = 1 xor 0 = 1

Sonuç;

S	E	F
1	1000 0110	0101 0110 0000 0000 0000 000

MIPS

MIPS'te otuziki tane 32bitlik register($f0 - $f31) içeren f*loating point coprocessor(yardımcı işlemci)* bulundurmaktadır. IEEE 754 single precision ve double precision formatlarında işlem yapmak için aşağıdaki kodlar registerları ile birlikte kullanılmalıdır.

- **Toplama, single (add. s), double (add . d)**
- **Çıkarma, single (sub. s), double (sub .d)**
- **Çarpma, single (mul. s), double (mul .d)**
- **Bölme, single (div. s), double (div .d)**
- **Karşılaştırma, single (c .x. s), double (c. x .d)**

x değişkeni equal(eq), not equal(neq), less than(lt), less than or equal(le), greater than(gt) veya greater than or equal(ge) olabilir.

- **branch, true (bc1t), false (bc1f)**

Floating-point comparison(karşılaştırma) ve Branch şarta bağlı olarak bir biti true veya false yapar.

lwc1 ve swc1komutları ile floating point single ve double precision registerlarına ($f*O,* $f1, $f2, ...)ayrılarak yazılmasını sağlayan komutlardır. Verilen MIPS kodu, 2 single precision sayıyı hafızadan alıp(load), topladıktan sonra hafızaya yollamaktadır(store).

```
lwc1   $f4,x($sp)      #     Load 32-bit F.P. number into F4
lwc1   $f6,y($sp)      #     Load 32-bit F.P. number into F6
add.s  $f2,$f4,$f6     #     F2 = F4 + F6 single precision
swc1   $f2, z($sp)     #     Store 32-bit F.P. number from F2
```

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2, . . . , $f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	FP add (double precision)
	FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 - $f6	FP sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 × $f6	FP multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	FP divide (double precision)
Data transfer	load word copr. 1	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	32-bit data to FP register
	store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1	32-bit data to memory
Condi-tional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

op(31:26):								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => *c* = f, rt(16:16) = 1 => *c* = t), rs(25:21):								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								

op(31:26) = 010001 (FlPt), (*f* above: 10000 => *f* = s, 10001 => *f* = d), funct(5:0):								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

Floating-Point C Programının MIPS Assembly kodunda derlenmesi

Örnek; Fahrenheit (F^o)'ı Celsius(C^o)'a çevirelim:

```
float f2c (float fahr)
        {return((5.0/9.0) * (fahr- 3 2.0));}
```

Sıcaklığın(Fahr) $f12'de, 5.0, 9.0 ve 32.0 sayılarının floating point değerlerin hafızada tutulan sabitler (sabitlere erişim pointer $gp) olduğunu varsayalım. Elde edilen sonucun(C^o) $ f0 registerına yazılması isteniliyorsa, MIPS assembly kodu:

f2c:

Hafızadan sabitlerin alınması

```
lwc1 $f16,const5($gp)  # $f 16 - 5.0 (5.0 In memory)
lwcl $f18,const9($gp)  # $f19 - 9.0 (9.0 in memory)
```

Fraction kısmını oluşturan 5.0/9.0

```
    div.s  $f16, $f16, $f18  # $fl6 = 5.0 / 9.0
```

32.0 sabitinin hafızadan alınıp, sıcaklıktan (fahr =$f12) çıkarılması:

```
    lwc1 $f18, const32($gp) # $f18 - 32.0
    sub.s $fl8, $f 12, $f18      # $f 18 = fahr - 32.0
```

Elde edilen 2 sonucun çarpılarak $f0 register'ına gönderilmesi

```
    mul.s $f0, $f16, $f18      # $f0 = (5/9>*<fahr - 32.0)
     jr $ra   #   return
```

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
char	—	lb, sb, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

Encodings of $\pm 2^{k+1-N}\, n$ into Binary Fields :

Number Type	**Sign** Bit	K+1 bit **Exponent**	Nth bit	N-1 bits of **Significand**
NaNs:	?	binary 111...111	1	binary 1xxx...xxx
SNaNs:	?	binary 111...111	1	nonzero binary 0xxx...xxx
Infinities:	±	binary 111...111	1	0
Normals:	±	$k-1+2^K$	1	nonnegative $n - 2^{N-1} < 2^{N-1}$
Subnormals:	±	0	0	positive $n < 2^{N-1}$
Zeros:	±	0	0	0