Chapter 3

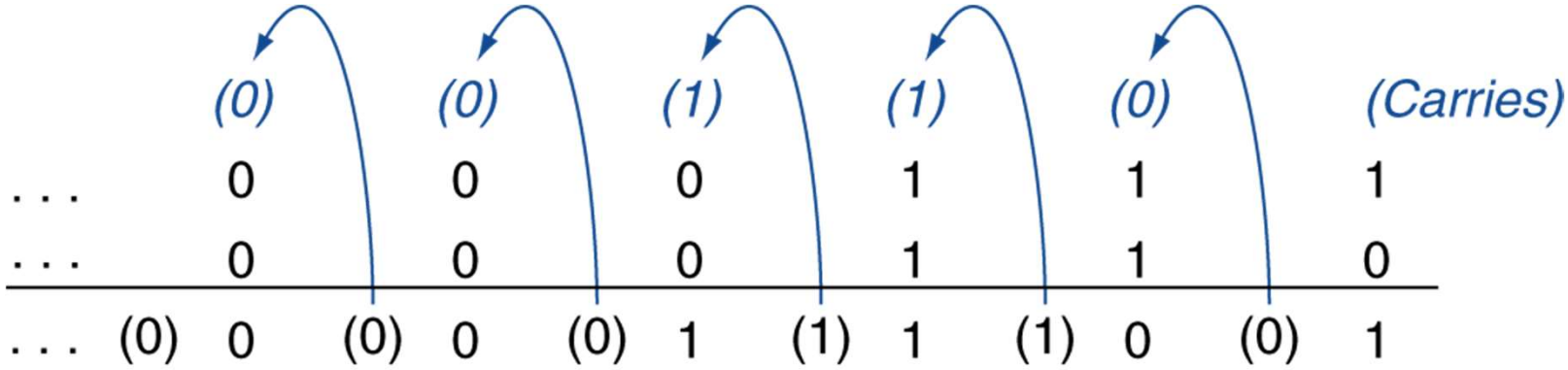
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from system control reg) instruction can retrieve EPC value, to return after corrective action

Multiplication

- Start with long-multiplication approach

multiplicand → 1000

multiplier → × 1001

1000

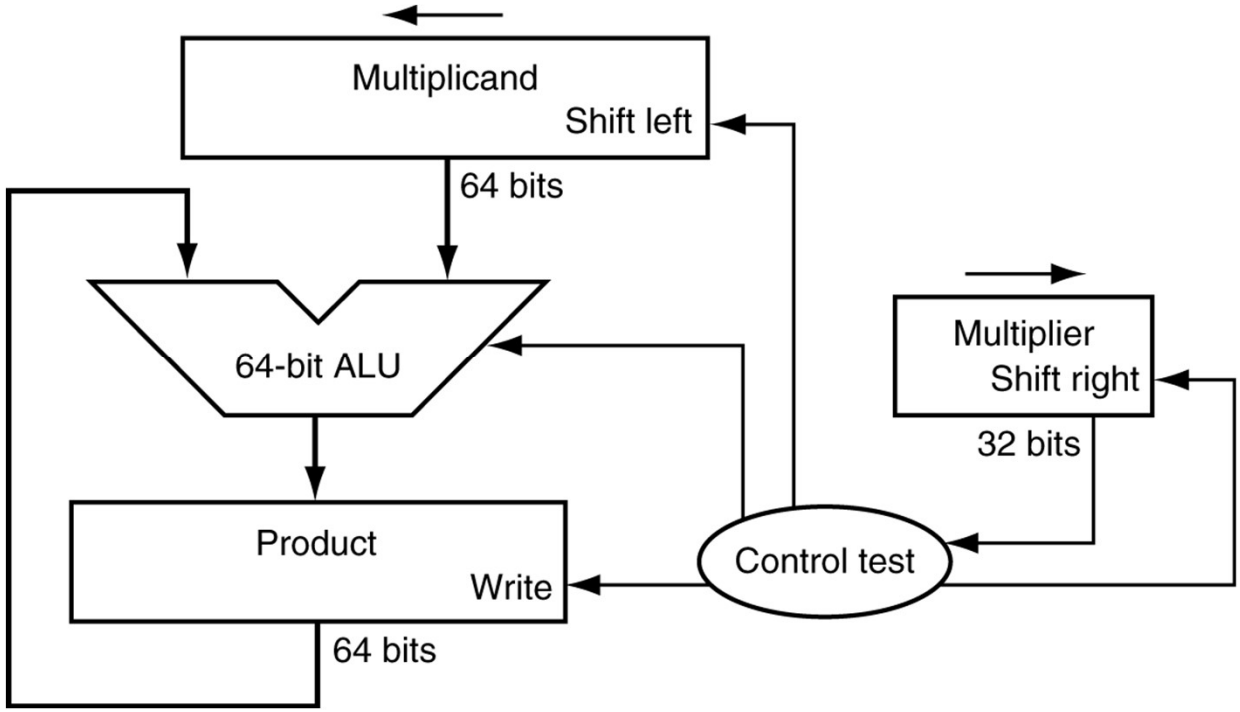
0000

0000

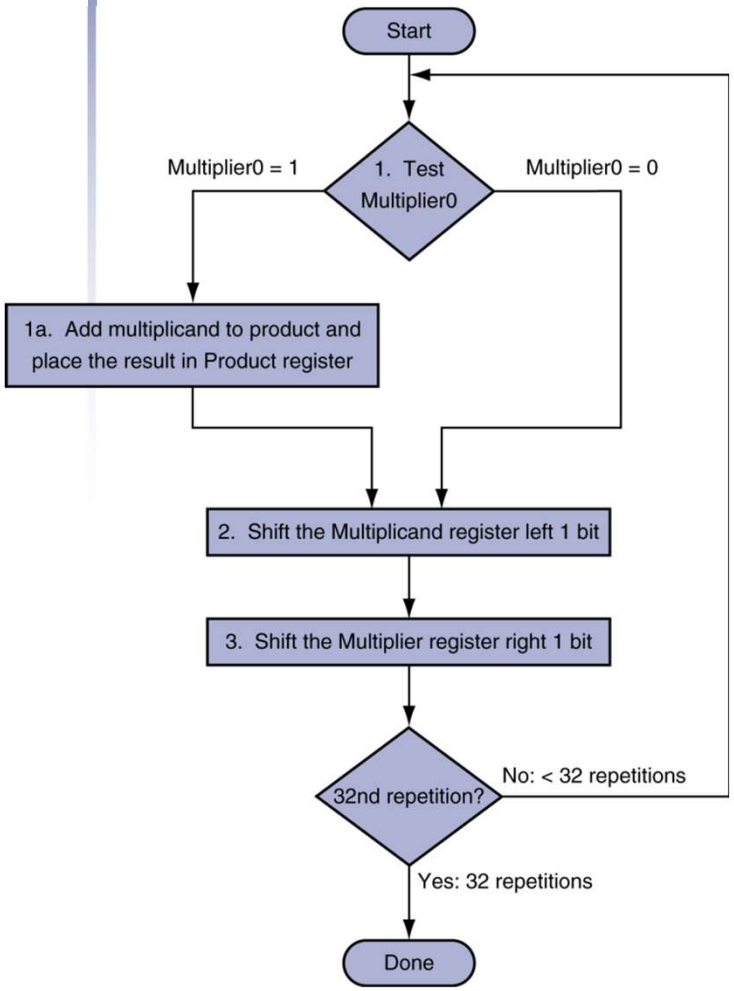
1000

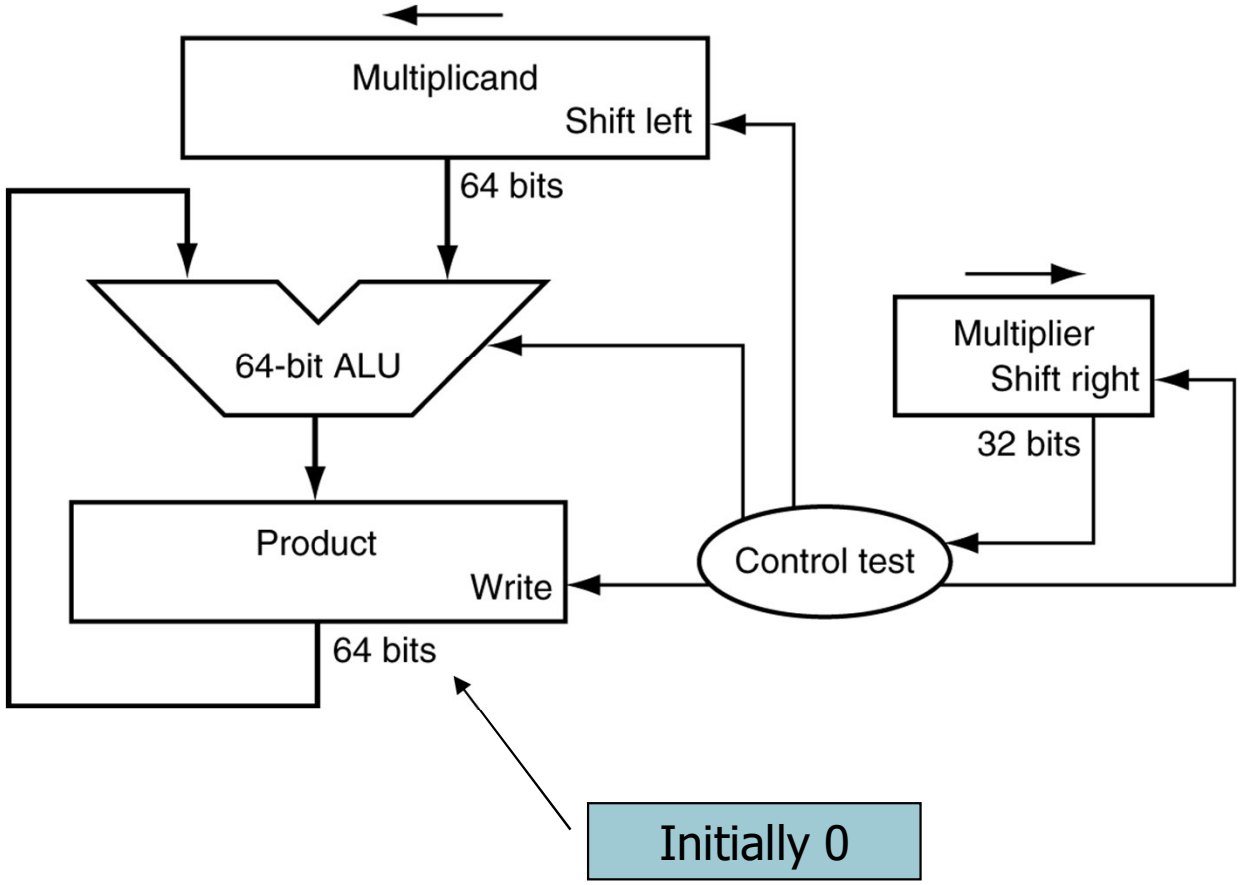
product → 1001000

Length of product is the sum of operand lengths



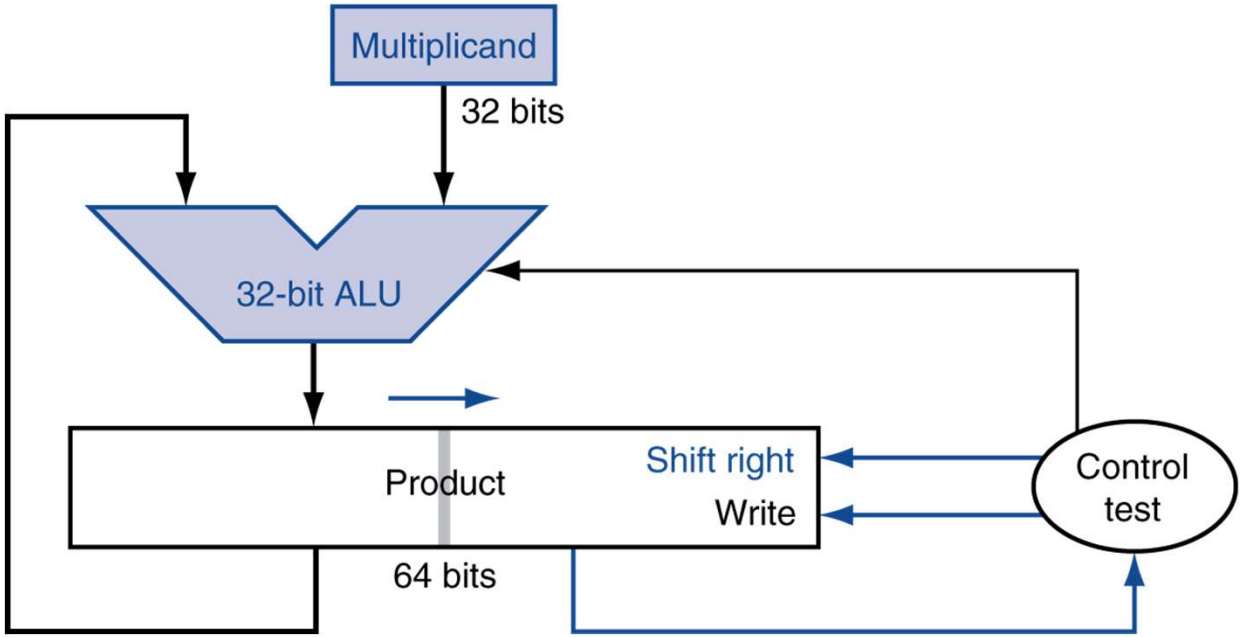
Multiplication Hardware





Optimized Multiplier

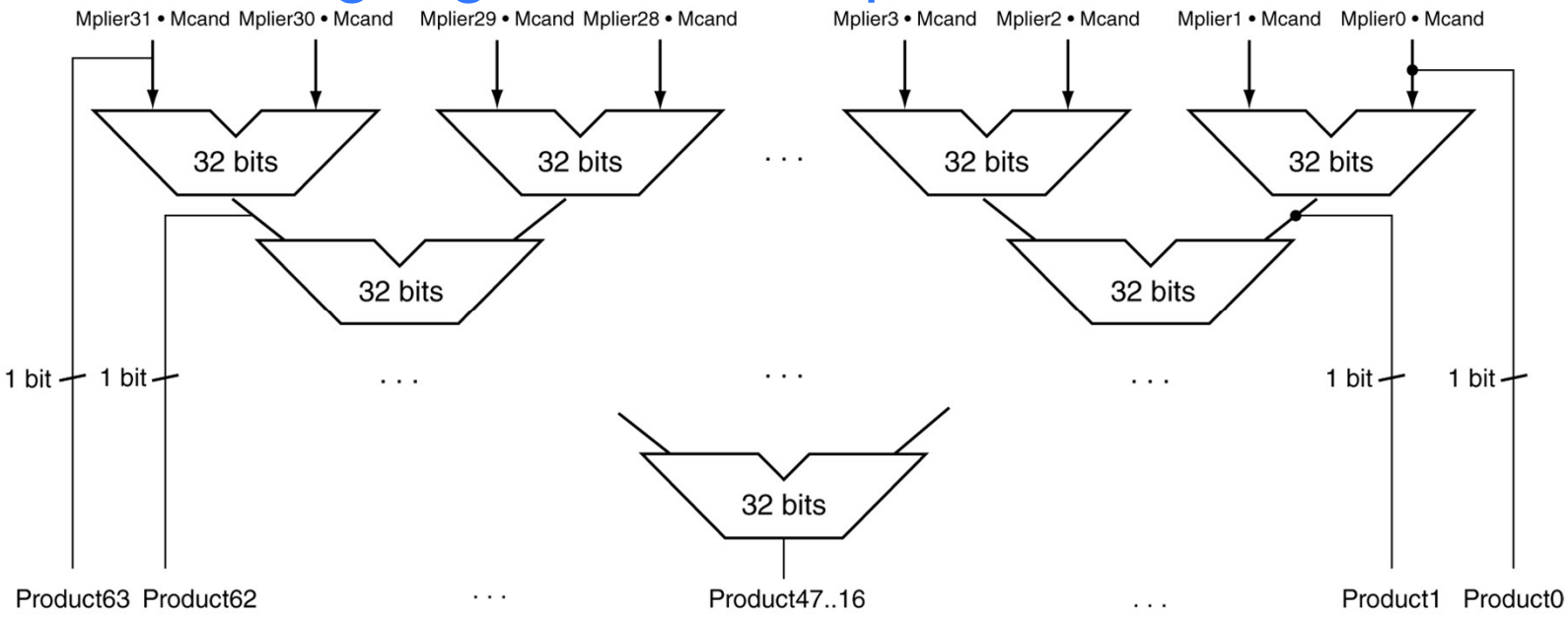
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff
  - Following Figure is incomplete

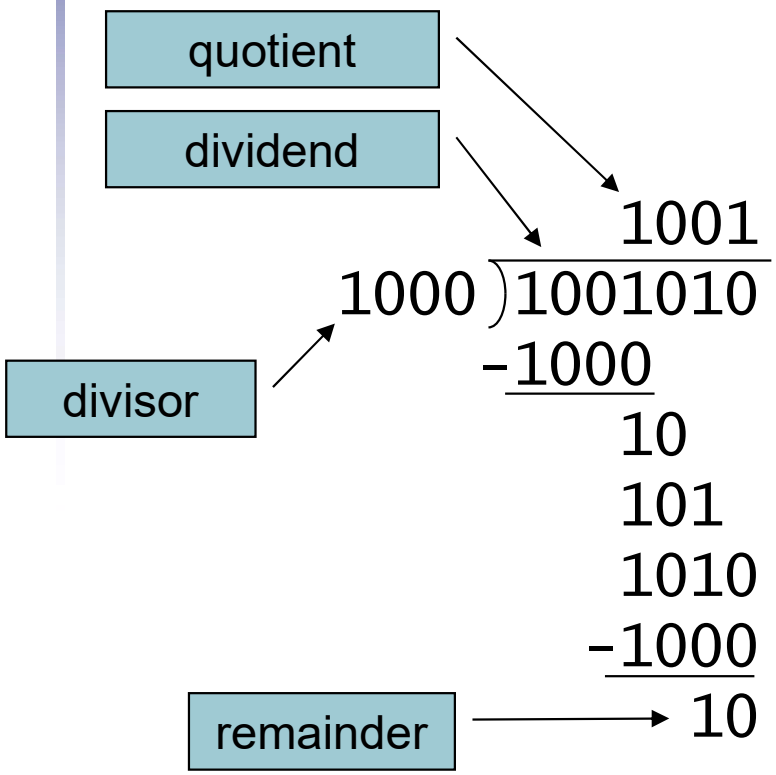


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt / multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

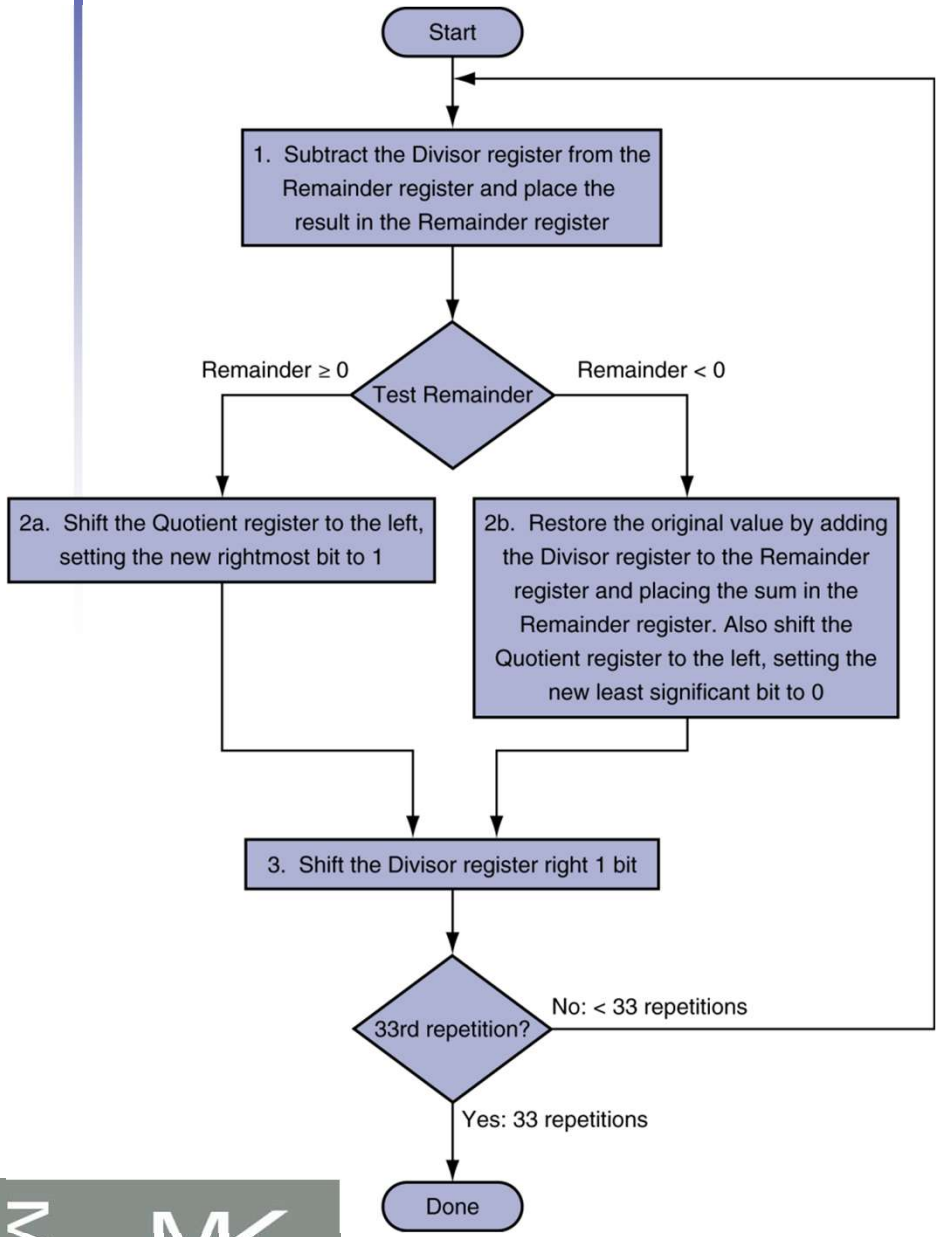
Division

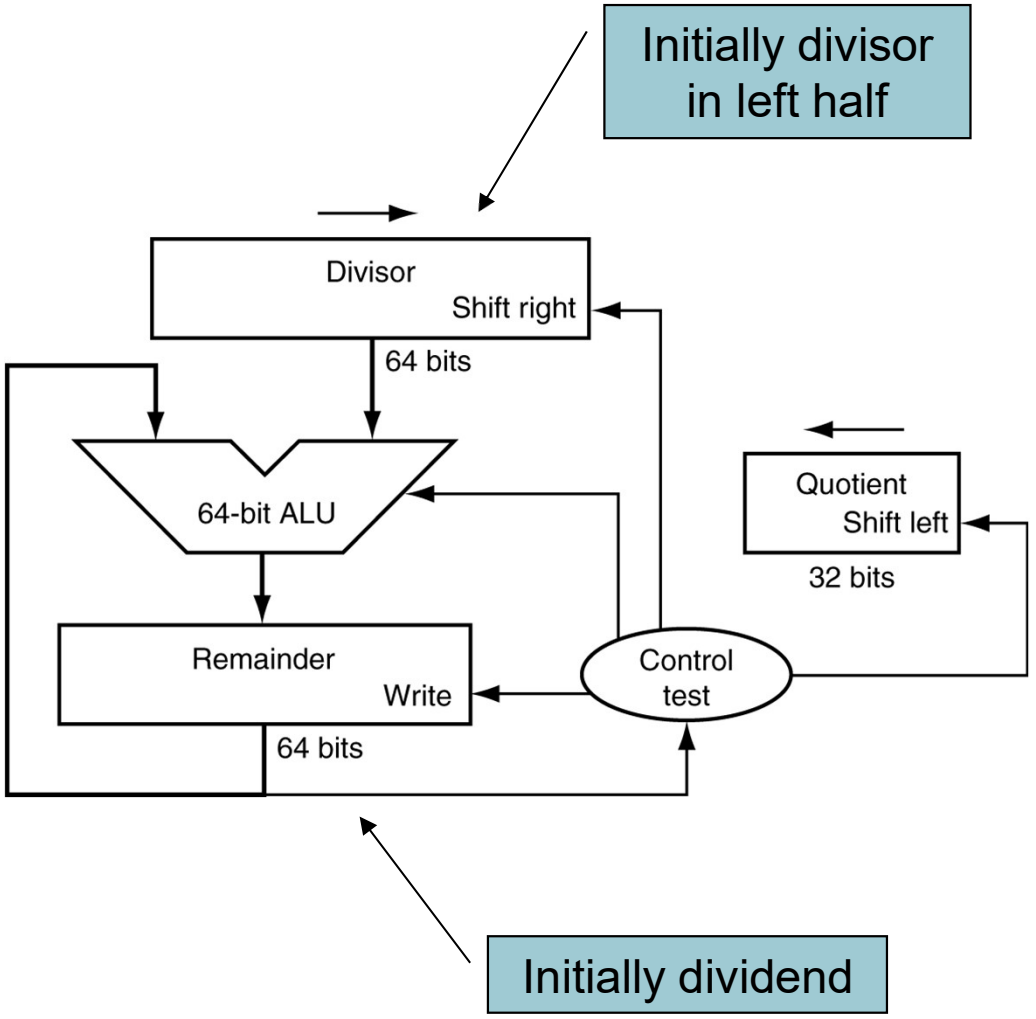


n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware

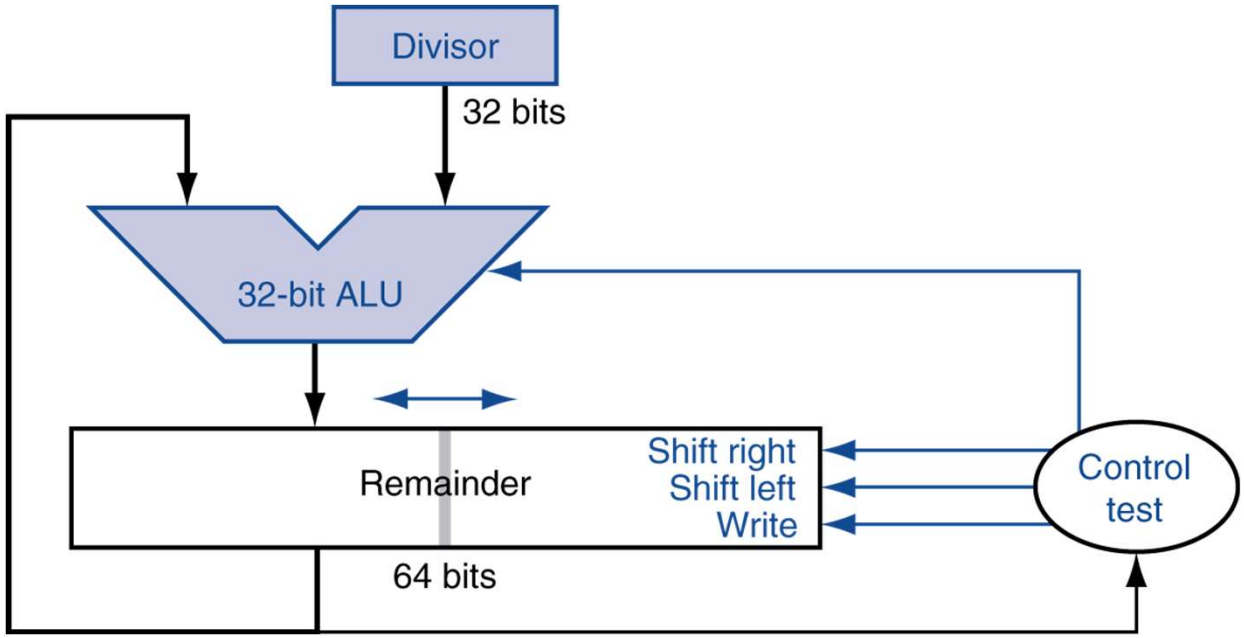




Division Example

Iteration	Step	Quotient	Divisor	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

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Optimized Divider

Iteration	Step	Divisor)	Remainder (R/ Q)	
0	Init	0010	0000	0111
1	Rem = Rem - Sub	0010	Negative!	0111
	Restore Rem Left Shift and Set 0	0010	0000 000**0**	0111 111**0**
2	Rem = Rem – Sub	0010	Negative!	1110
	Restore Rem Left Shift and Set 0	0010	0000 00**01**	1110 11**00**
3	Rem = Rem – Sub	0010	Negative!	1100
	Restore Rem Left Shift and Set 0	0010	0001 0**011**	1100 1**000**
4	Rem = Rem – Sub	0010	0001	1000
	Left Shift and Set 1	0010	001**1**	000**1**
5	Rem = Rem – Sub	0010	0001	0001
	Left Shift and Set 1 (special shift)	0010	0001	001**1**

Optimized Multiplier

Iteration	Step	Multiplicand	Result / Multiplier	
0	Init	0111	0000	0101
1	LSB =1, Add	0111	0111	0101
	RS	0111	0011	1010
2	LSB =0, Pass	0111	0011	1010
	RS	0111	0001	1101
3	LSB =1, Add	0111	1000	1101
	RS	0111	0100	0110
4	LSB =0, Pass	0111	0100	0110
	RS	0111	0010	0011

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt / divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result
    - Q in LO, R in HI

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and double in C

scientific notation → decimal point এর বামনে যাই থাকুক

যদি decimal point এর বামনে একটাই digit থাকে (non-zero)

↳ normalized standard form

$\pm 1.(\ldots)_2 \times (2)^{\square}$ ← এটাও বাইনারীতে থাকবে

① বাইনারীতে না লিখলেও হবে

↳ বাইনারিতে থাকবে

exponent term যদিও hardware এ binary তে থাকে

আমরা decimal এ লিখে কাজ করবো,

binary representation লিখতে বলা হলে → convert to binary.

$(1\ 0\ 1)_2 \rightarrow 1\times2^2 + 0\times2^1 + 1\times2^0$

① ↳ এই 2 গুলো থাকে না

$\pm 1 . (\ldots\ldots)_2 \times 2^{\square}$

যে part গুলো থাকাই লাগবে

1. fraction → implicit 1 / floting point এর পরের part টুকু
2. exponent
3. sign

2 ↗ এই power এর উপর depend করে point কে যেকোনো জায়গায় নিতে পারি।

— float করছে

তাই floating point বলা হয়।

যদি 16 bit architecture হয় —

MIPS — 32 bit architecture

↳ ALU 32 bit — operand গুলো 32 bit এর হতে হবে → 16bit → 16bit

→ register 32 bit

$\pm 1.(\ldots\ldots)$

↓

sign representation এ

1 bit লাগবে। (32-1) = 31 bit বাকি

fraction, exponent এ কতটুকু দিবো fixed করা যায়?

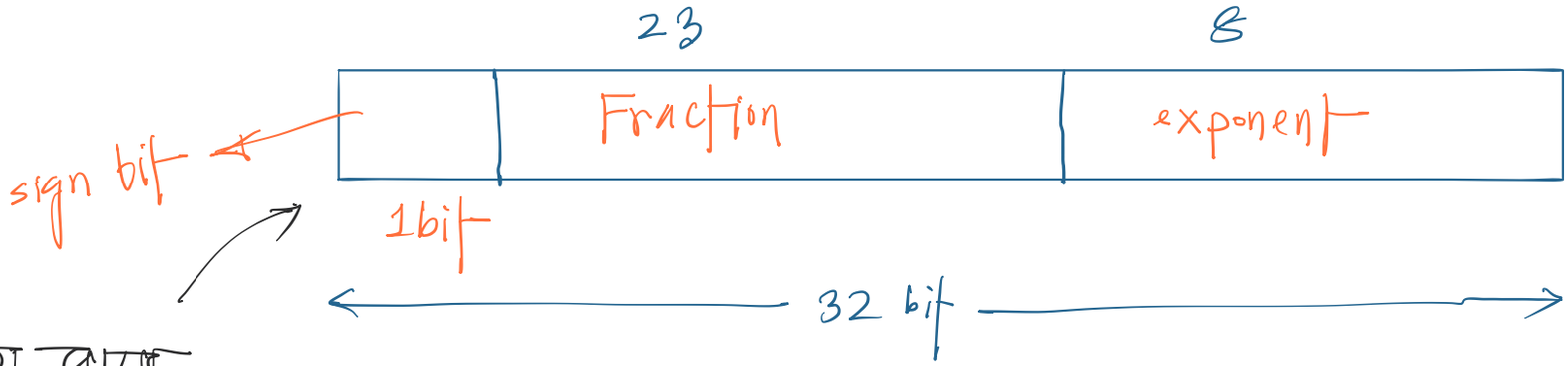
↳ trade off করা লাগে। (Good design needs good compromises)

exponent বাড়ালে (10) ⟶ অনেক বড় একটা number রাখতে পারবো $2^{10} = 1024$

fraction " ⟶ precision বাড়বে

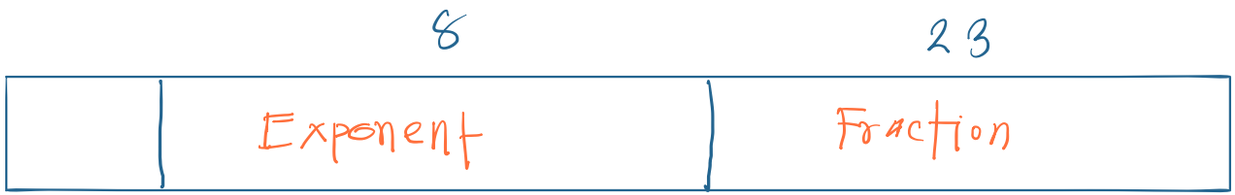
{ 8 bit exponent
  23 bit fraction এ

represent করা হয়



এটায় কোনো extra facilities পাচ্ছি না।

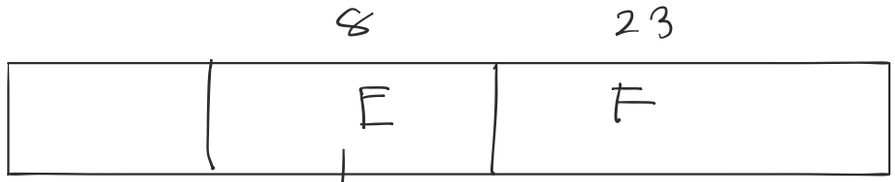
↳ fraction সামনে রাখলে compare করতে পারবো না। suppose fraction বড় but exponent কম। But comparison operation অনেক হবে। so, ↳ comparison faster করতে



comparison → আগে sign bit mismatch check

মিলে গেলে exponent check (high probability যে এখানে mismatch থাকবে

কিছু ,, ,, fraction check



binary to decimal conversion

$F \times 2^{-1} + F_2 \times 2^{-2}$

problem হবে — যদি 0 আসে

আমরা কিছু combination fix করে দিবো

exponent = 0

Fraction full part = 0

(less occurring case)

এমন scenario তে 1 আর add হবে না।

↳ make the common case first (1 add হওয়া common), এটা rare case.

exponent can also be positive or negative.

↳ exponent এর প্রথম bit sign bit রাখবো।

$-1 \rightarrow$ 1 1 1 1 1 1 1 1

$+1 \rightarrow$ 0 0 0 0 0 0 0 1

bit by bit compare করা ছাড়া উপায় নেই। তখন -1 বড় আসবে।

∴ 2's complement করা যাবে না। (exponent comparison ও আটকে যাবে)

soln: bias calculate: $\underbrace{2^{n-1} - 1}$

8 bit এর জন্য $= 2^{8-1} - 1$

$= 127$

127 যোগ করে সবগুলো scale up করবো। negative গুলো positive হবে।

$-1 \longrightarrow 127 + (-1) = 126$

↓

binary তে represent

$1 \longrightarrow 128 \longrightarrow$

এরপর bit by bit compare.

∴ বড় value গুলো always বড়-ই থাকবে।

∴ exponent এর যে value টা পাবো সেটা −127 হলো power

$2^{\boxed{\text{power}}}$

decimal এ { precision, range } কত হবে?

binary ⓑ { precision 23 bit + implicit 1

= 24bit

range → -127 to 127 $-(2^7-1)$ to (2^7-1)

decimal ⓓ $2^{127} = 10^{\square}$ → 38.... (range)

$2^{23} = 10^{\square}$

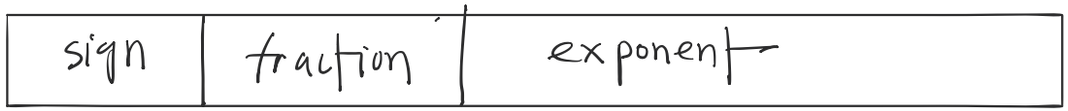
6.... → precision

float → 4 byte

double → 8 byte (64 bit)

sign	fraction	exponent

1 + 52 + 11



1 + 52 + 11

precision and range both in binary and decimal ?

binary { precision = 52
range = −1023 to 1023

decimal:

range $2^{1023} = 10^x$

$x = 307.95$

precision $2^{52} = 10^x$

$x = 15.65$

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

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)

Double-Precision Range

52
11

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

1 + fraction
0

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - A float has 23 bits of mantissa, and 2^23 is 8,388,608. 23 bits let you store all 6 digit numbers or lower, and most of the 7 digit numbers. This means that floating point numbers have between 6 and 7 digits of precision, regardless of exponent.
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

$-(0.75)_{10}$

- প্রথম কাজ বাইনারীতে নেওয়া। (২ দিয়ে গুন)
  → কতক্ষণ করবো?
  — যত bit number তত্তবার

$= -(0.11)_2$

- ২য় কাজ normalized form এ নেওয়া।

$= -1.1 \times 2^{(-1)}$ → bias term add

$-1 + 127 = 126$

→ বাইনারীতে represent করা লাগবে।

$= -1.1 \times 2^{126}$

exp : 0 1 1 1 1 1 1 0

1 0 1 1 1 1 1 0 1 ...

Fraction: 1 0 0 0 0 ....

MSB থেকে শুরু হয়।

#

1.0×2^0

exp = 127

= 1111111

1×2^{127}

fraction = 0000 ...

exp = 1111111

S = 0

0 কে floating point এ express করতে পারি না।

implicit 1 নাই।

1 কে করতে পারি কারণ exponent এ 127 আসবে।

0 না।

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111101000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100000...00

  - S = 1
  - Fraction = $01000...00_2$
  - Exponent = 10000001_2 = 129

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

$(.01)_2 = 2^{-2}$

$= (0.25)_{10}$

addition :

0.75×10^2

0.004×10^1 ① exponent টা ছোট। সমান করা লাগবে।

fraction টায় right shift.

0.0004×10^2 exponent সমান হলো।

$$\begin{array}{rl} & 0.7500 \times 10^2 \\ + & 0.0004 \times 10^2 \\ \hline & 0.7504 \times 10^2 \end{array}$$

$= 7.504 \times 10^1$ normalized form এ নিতে হবে।

binary তেও same. But implicit 1 কে সহ left shift / right shift করা লাগবে। exponent দুইটা সমান করতে।

1.1

right shift = 0.11

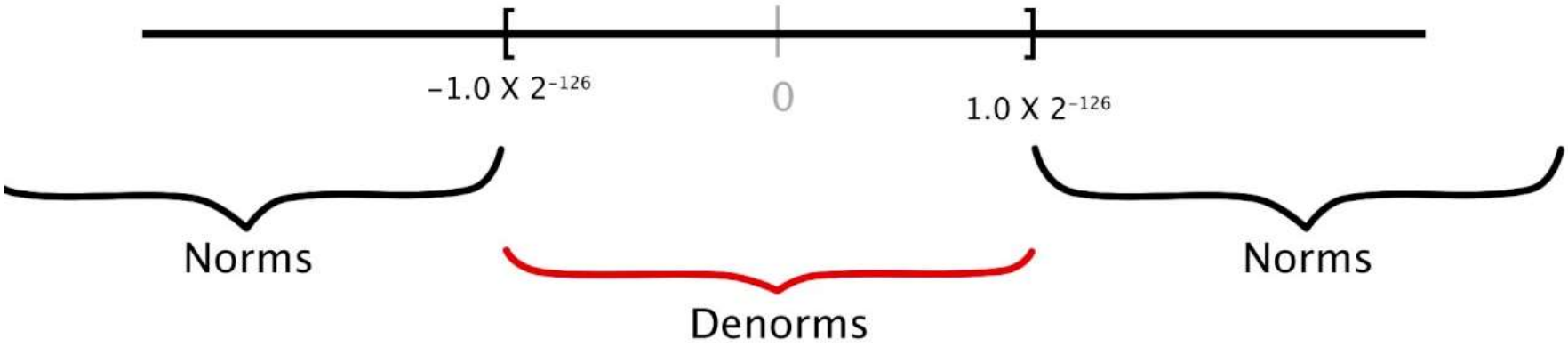
implicit 1 টা ভিতরে

∴ implicit value এখন 0।

Denormal Numbers

Denormalized Numbers $\pm 0.B \times 2^e$

Every denormalized number is smaller in absolute value than every normalized number.



Denormal Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision

- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

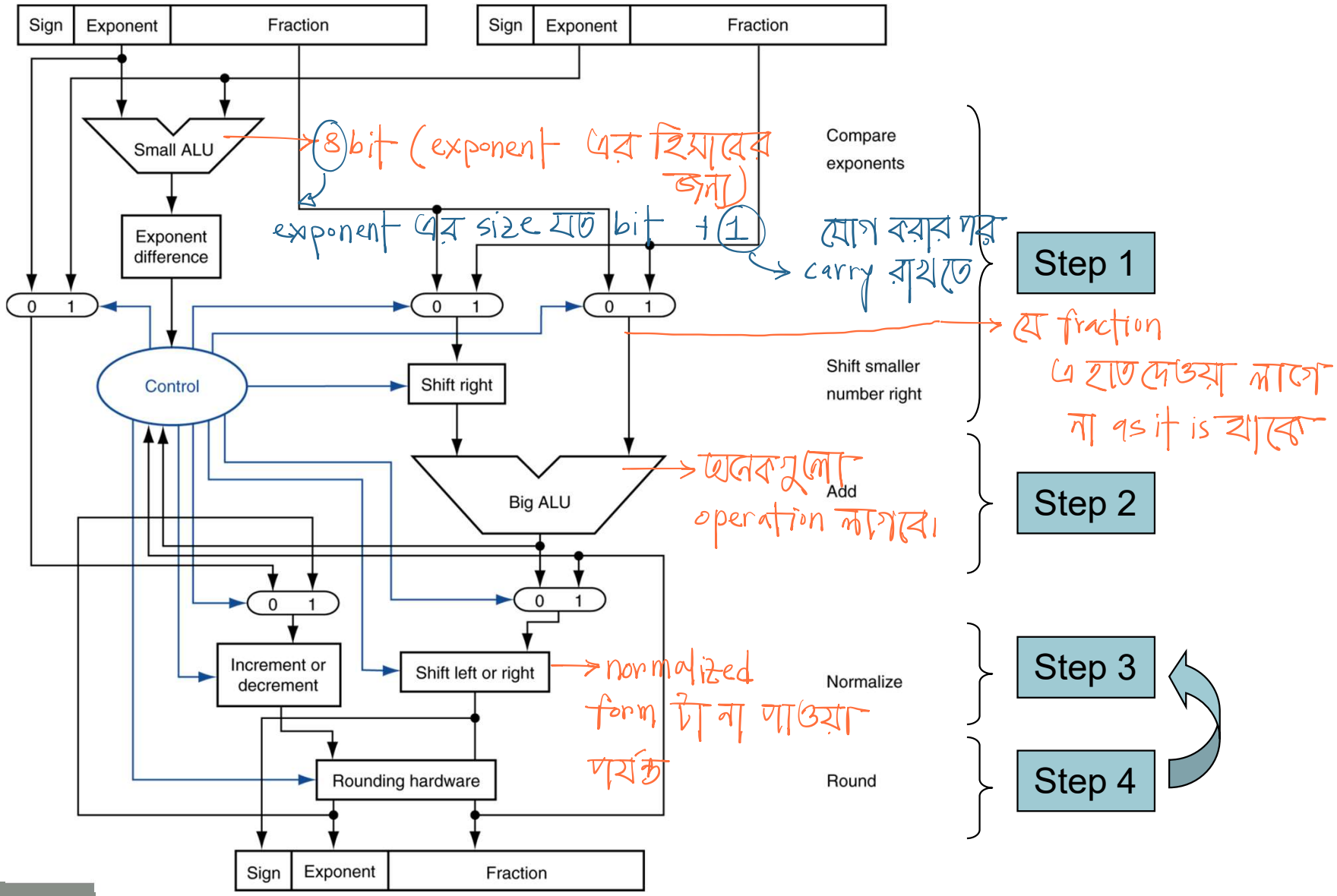
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Rounding Bits

- Guard and round digits, and sticky bit
- When computing result, assume there are several extra digits available for shifting and computation. This improves accuracy of computation.
- Guard digit: first extra digit/bit to the right of mantissa -- used for rounding addition results
- Round digit: second extra digit/bit to the right of mantissa -- used for rounding multiplication results
- Sticky bit: third extra digit/bit to the right of mantissa – used for resolving ties such as 0.50...00 vs. 0.50...01

Rounding Bits

- Rounding using Guard and round digits, and sticky bit

G	R	S	Action
0	0	0	Truncate
0	0	1	Truncate
0	1	0	Truncate
0	1	1	Truncate
1	0	0	Round to Even
1	0	1	Round Up
1	1	0	Round Up
1	1	1	Round Up

$1.100GRS$

$1.10001 = 1.53125$
$1.10010 = 1.56250$
$1.10011 = 1.59375$

Multiplication:

```
multiplicand      1 0 0 0
multiplier        1 0 0 1
                ---------
                  1 0 0 0
                0 0 0 0
              0 0 0 0
            1 0 0 0
            ---------------
            1 0 0 1 0 0 0
```

Intermediate value (store করা লাগবে memory তে) finally add করার সময় memory থেকে আনা লাগবে। 2টা করে আনতে পারবো।

প্রতি memory operation এ 2টা load, একটা store

→ memory operation অনেক costly.

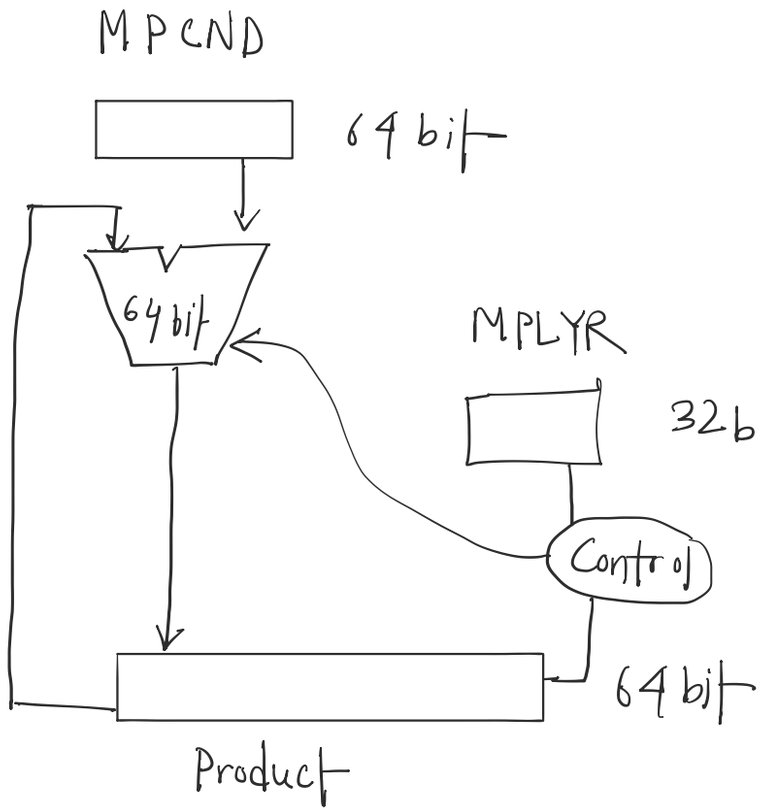
current bit

multiplier এ 1 আসলে add হবে (current multiplicand + product)

0 " add হবে না।

প্রতি step এ multiplier একবার right shift করবো

multiplicand left shift করবো।



left shift করছি।
কোনো part বাদ দিতে পারবো না।

righmost bit ফেলে দিলেও সমস্যা নাই

option 1 multiplier এর right most bit দেখে add করবো কিনা product এ যোগ করা decide ⟶ implement করা easy. control দেখে decide

option 2 ALU কে কাজ করতে দিবো না।

↳ but সে আগের result ই দেখাবে।

0000	1011

0101

প্রতিবার add হবে, কিন্তু multiplier এর value 1 হলে addition এর result store হবে।

0000	0000

00010110 left shift

sum হবে but 0 থাকায় store হবে না।

(5)

0010

right shift

0000	1011

00101100
+ 00001011

Sum = 00110111

←

0 01011100

S2

→

0001

0011 0111

←

0101 1000

S3

→

0000

↳ কাজ শেষ না। ৪ বার
করা লাগবে।

0011 0111

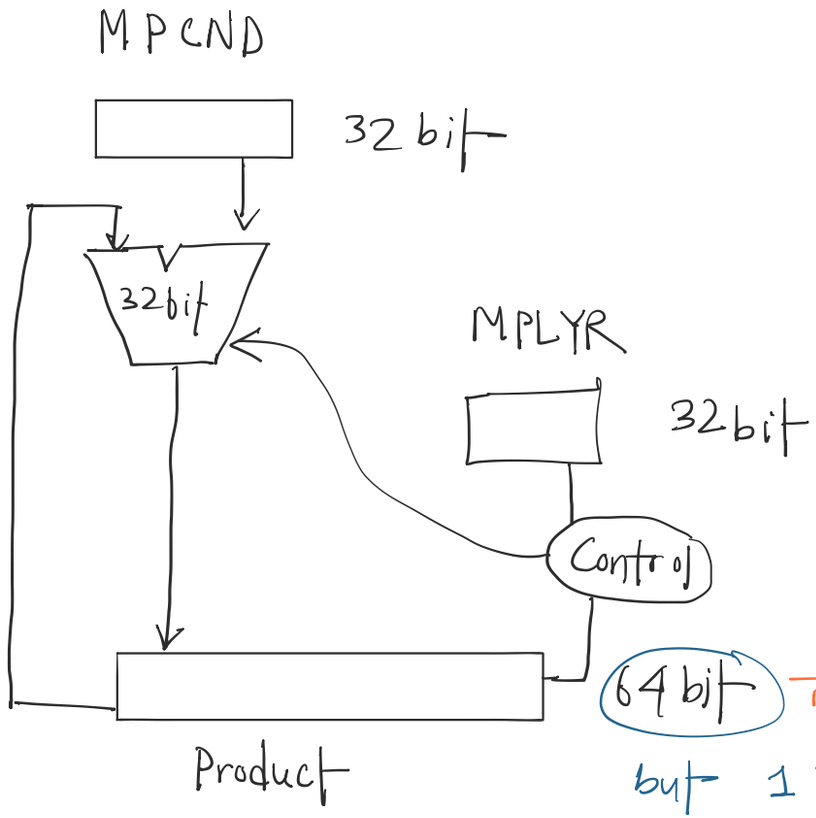
1011000

0000

0011 0111

S4 → counter বলবে
4 steps done.
multiplication
শেষ

32 bit processor এর মধ্যে কখনোই 64 bit ALU পাবো না।



দিয়ে handle করে
লাগবেই। MIPS 2টা special register
but 1 টা 64 না বরং 2 টা 32 bit.
High ↔ Low

→ Reduced Instruction set..

power consumption ← RISC → instruction set reduce (hardware...) ARM architecture
খুবই কম
• Apple vision

CISC → কম resource এ অনেক বাজ
instruction set অনেক powerful.

Complex instruction set..

RISC

↓

hardware বাড়ায়।

↳ power usage কমায়

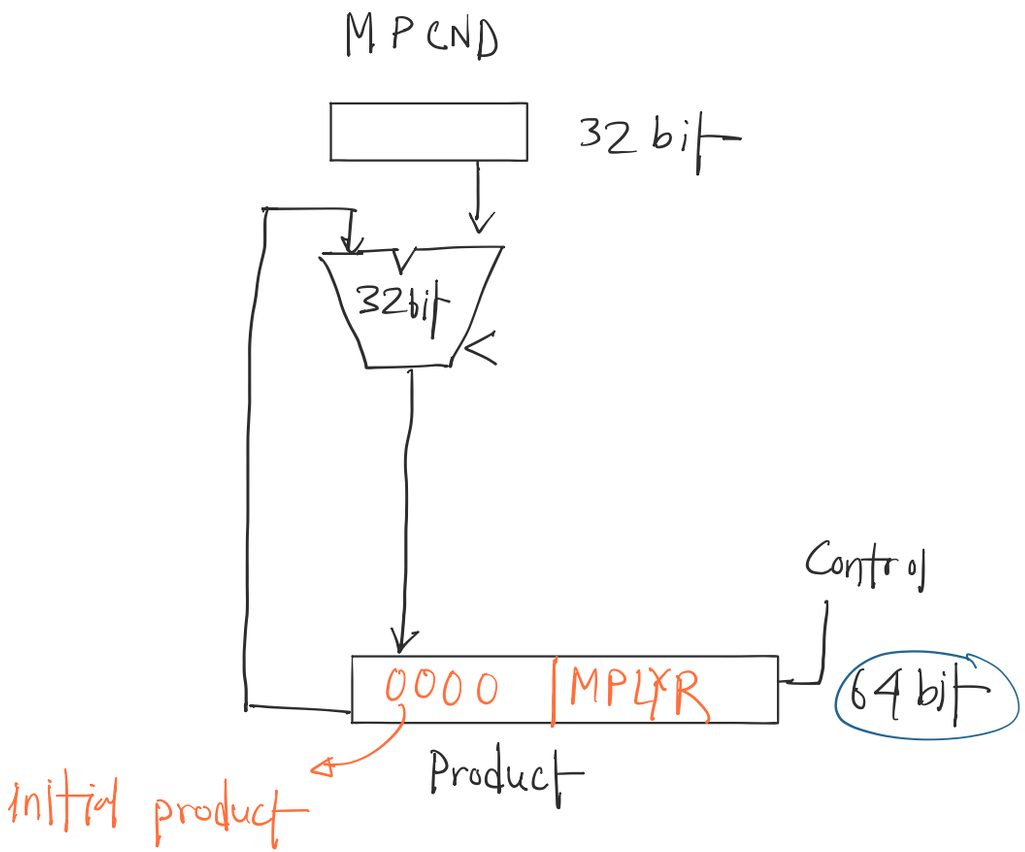
পরীক্ষায় আসবে

যে তে multiplication / division থাকবে

– ৪ bit থাকতে পারে

→ MPCND left shift করতে পারবো না। তাহলে 32 bit exceed করে। Soln:

product কে right shift করবো।



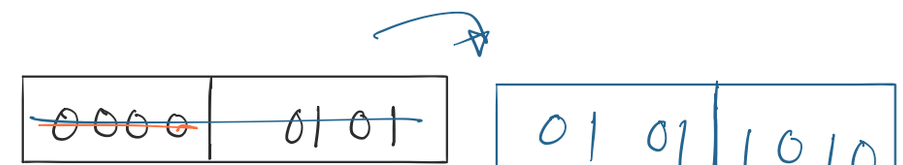
প্রতিবার multiplicand এর সাথে product এর left half add.

↳ update করবো কি না multiplier থেকে decide

→ update হবে শুধু left half.

1011

S1



1011

update

then

right shift

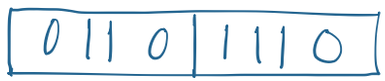
S2

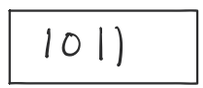
product এ নিবো না and right shift

S3

right shift

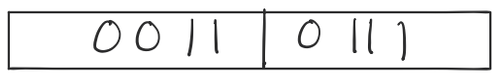
add
update
right shift





যতক্ষণ multiplier exhaust হচ্ছে না ততক্ষণ করবো।

S4



product

numerical example বই এ যেটা আছে এই hardware এর নীচে আছে সেটা এটার example না।

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

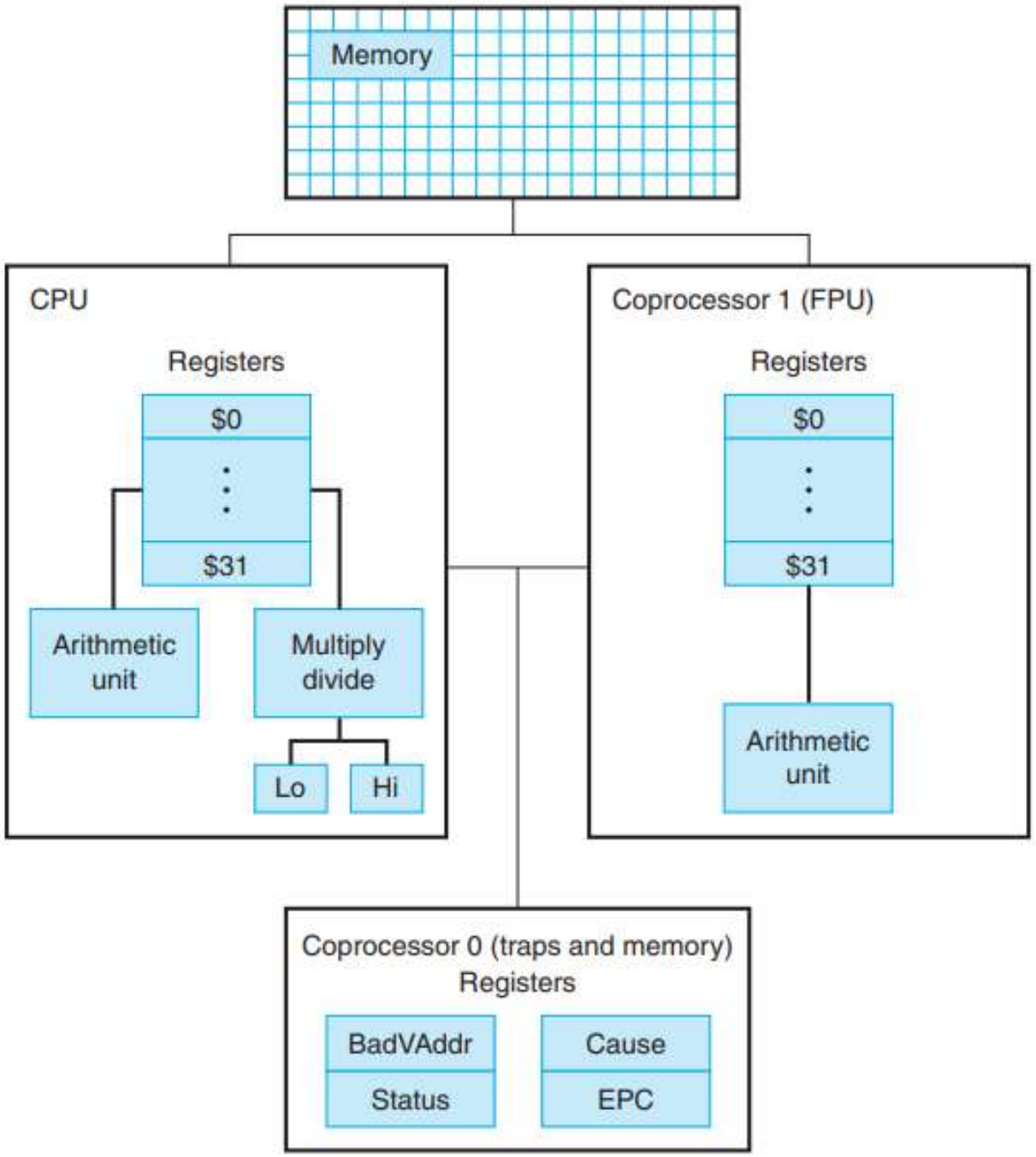
- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

Associativity of FP Addition

- Associativity: a+(b+c) = (a+b) + c
- Is FP addition associative?
  - No

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`


Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Cause
Status
EPC

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

Single and Double Precision operations

- A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.

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

- What if add.d was used?

Floating-Point Instructions in MIPS

- Floating-point *addition, single* (add.s) and *addition, double* (add.d)
  - e.g., add.s $f0, $f4, $f6 # $f2 = $f4 + $f6
- Floating-point *subtraction, single* (sub.s) and *subtraction, double* (sub.d)
  - e.g., sub.d $f2, $f4, $f6 # $f2 = $f4 - $f6
- Floating-point *multiplication, single* (mul.s) and *multiplication, double* (mul.d)
  - e.g., mul.s $f2, $f4, $f6 # $f2 = $f4 X $f6
- Floating-point *division, single* (div.s) and *division, double* (div.d)
  - e.g., div.d $f2, $f4, $f6 # $f2 = $f4 / $f6

Floating-Point Instructions in MIPS

- Floating-point *comparison, single* (c.x.s) and *comparison, double* (c.x.d),
  where x may be *equal* (eq), *not equal* (neq), *less than* (lt), *less than or equal*
  (le), *greater than* (gt), or *greater than or equal* (ge)
  - e.g., c.lt.s $f2, $f4 # if ($f2 < $f4) cond = 1; else cond = 0
- Floating-point *branch, true* (bc1t) and *branch, false* (bc1f)
  - e.g., bclt 25 # if (cond == 1) go to PC + 4 + 100

Floating-Point Instructions in MIPS

```
if (i==j)
    f = g + h;
else
    f = g – h;
```

```
        c.eq.s $f2, $f4
        bc1f Else          #go to Else if i ≠ j
        add.s $f6,$f8,$f10 #f = g + h
        j Exit
Else:   sub.s $f6,$f8,$f10 #f = g – h
Exit:
```

MIPS FP Operands

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 FP Operations

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 FP Instruction Format

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

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

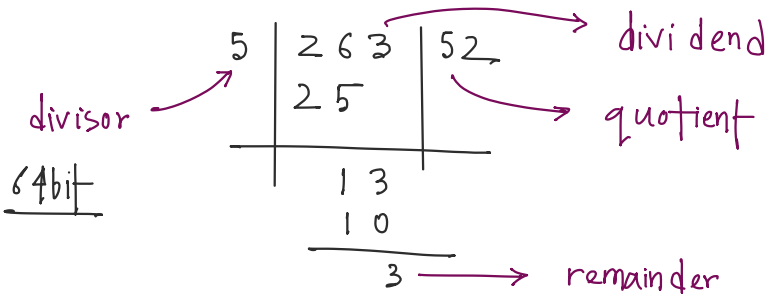
Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

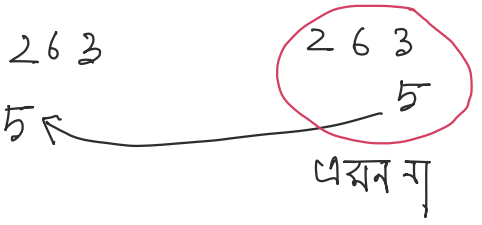
- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

29.10.24



subtraction হচ্ছে left এ

most of the time divisor ছোট হবে।



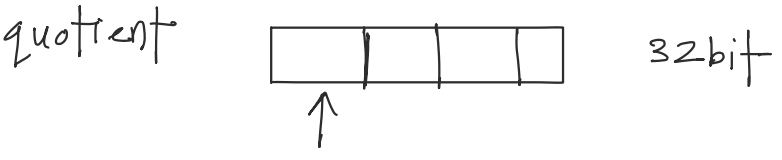
5 কে full shift করা লাগবে।

আসলে full shift না করে 1 bit করে shift লাগে।

but এটা আমরা করবো না।

divisor 64 bit

dividend 64 bit। নাহলে subtract করা যাবে না।



প্রথমটা এখানে আসবে।

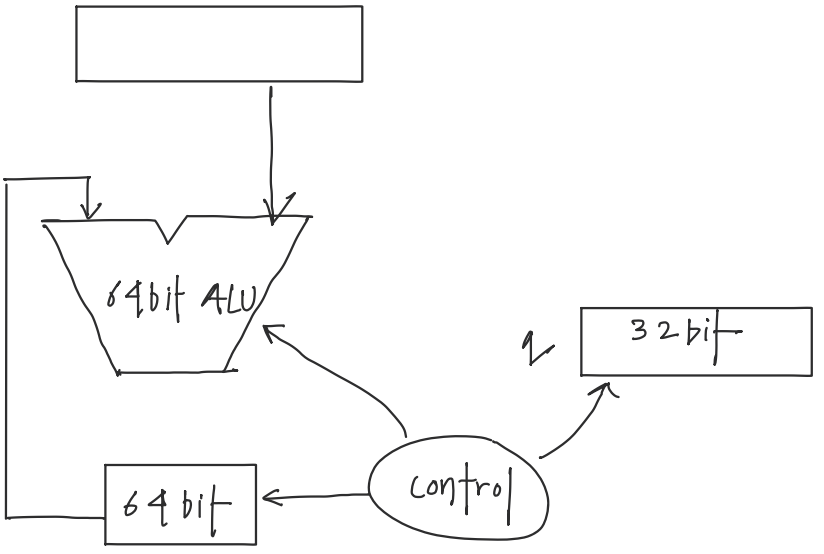
এরপর right এ

option 1: এখন কোথায় বসবে সেই count রাখার জন্য আলাদা register

option 2: last এ insert then shift.

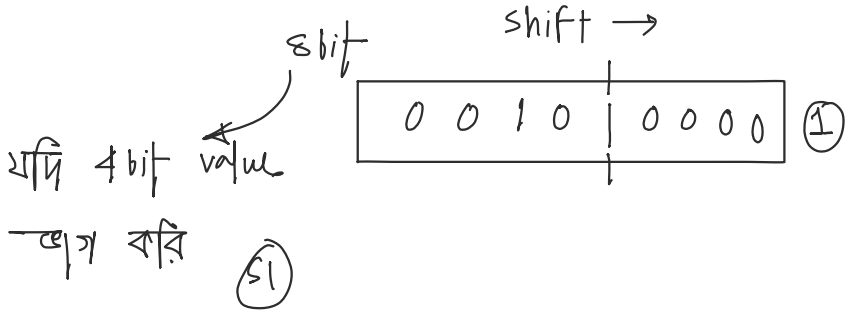
dividend কমে কমে reminder হবে। তাই reminder এর জন্য আলাদা register লাগে না।

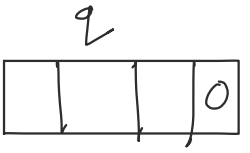
Divisor 64 bit

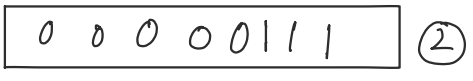


way 1 → 1টা extra register রাখবো যেখানে subtract এর result থাকবে। sign দেখে...

way 2 → একটা extra operation লাগবে। বইতে এটা follow.





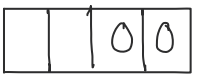


② - ① হয়
এখানে করা যাচ্ছে না
So quotient এ 0।
1 ঘর shift

S2

0 0 0 1 | 0 0 0 0

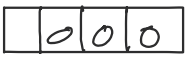
0 0 0 0 | 0 1 1 1



S3

0 0 0 0 | 0 0 0

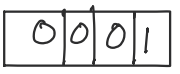
0 0 0 0 0 | 1 1 1



S4

0 0 0 0 0 | 1 1

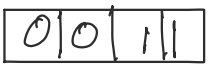
0 0 0 0 0 | 1 1 1



S5

0 0 0 0 0 0 | 0 1

0 0 0 0 0 0 | 1 1



right এ 1 আসবে then left shift.

00000111
0000 0100
00000011

n+1 step লাগলো → full shift করায় (32 shift লাগে)

n step এ হতো → 1 bit কম shift করা হলো

00 11

n-1 shift দিয়ে করা

0 0 0 1 | 1 0 0 0

00 11 নিয়ে shift করা লাগবে 31 shift

algorithm এ লাগে 32 shift.

∴ total 13 shift

আগে ছিল (32) ? 33?

full shift ৬

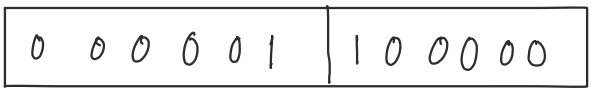
divisor 0 00011

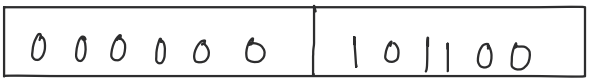
dividend 10 11 00

000011 | 000000

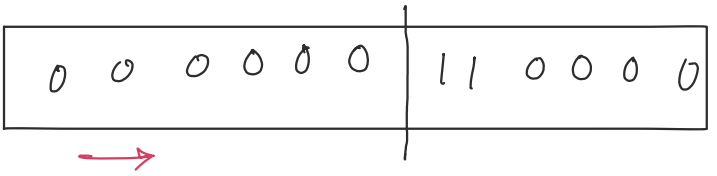
000000 | 101100

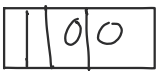
S1

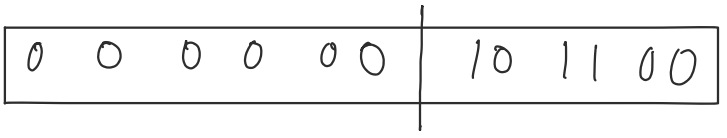




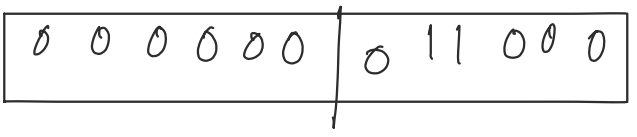
S2

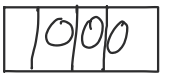


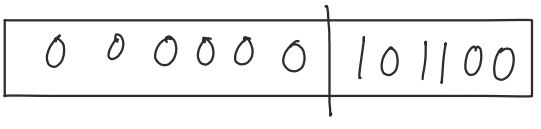




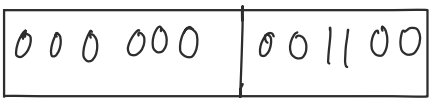
S3

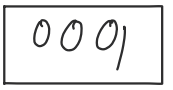


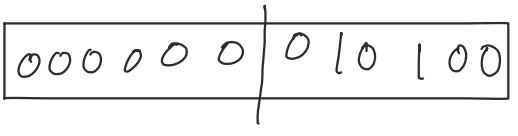




S4







(sub)

101100

− 011000

010100

010100
001100
001000

S5

0 0 0 0 0 0 | 0 0 0 | 1 1 0

00011

0 0 0 0 0 0 | 0 0 1 0 | 0 0

S6

0 00000 | 0 0 0 0 | 11

000111

000 000 | 0 0 0 1 0

S7

0 00000 | 00011

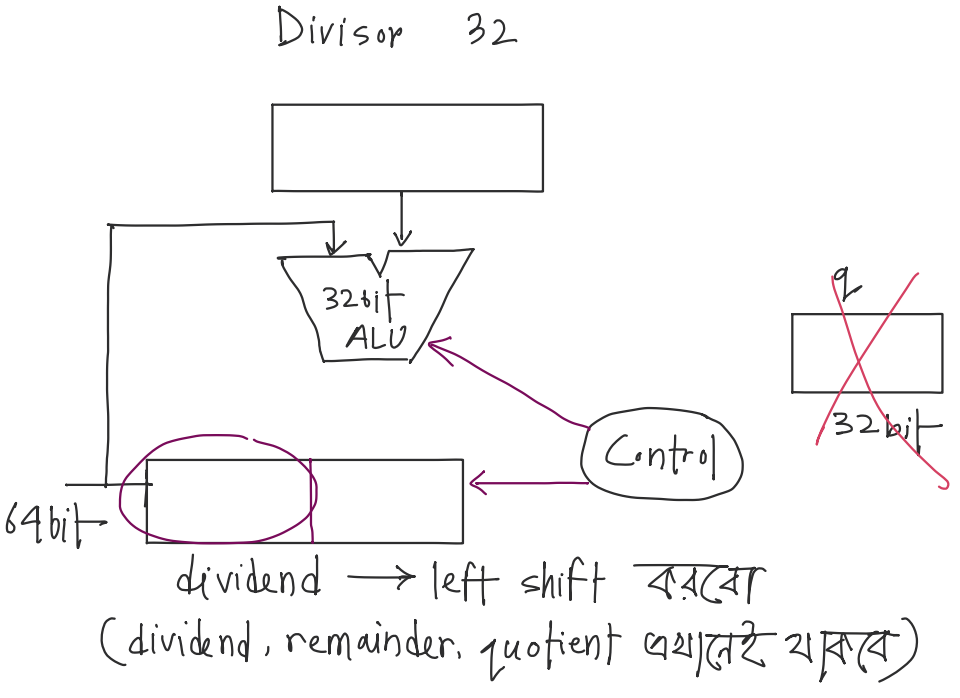
001110

quotient

000000 | 000010

remainder

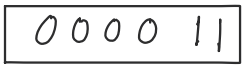
এই hardware real life এ feasible না কারণ 64 bit ALU.



dividend → left shift করবো

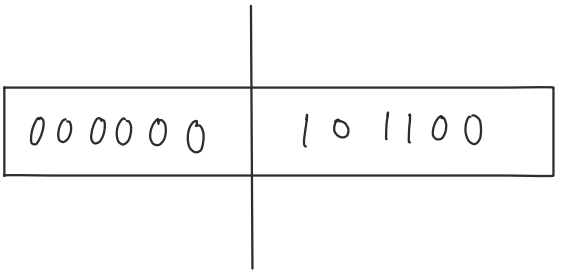
(dividend, remainder, quotient একসাথেই যাবে)

quotient এ যে value টা থাকার কথা left shift এ যেই value দিয়ে fillup করবো। তাহলে quotient এর register লাগছে না।

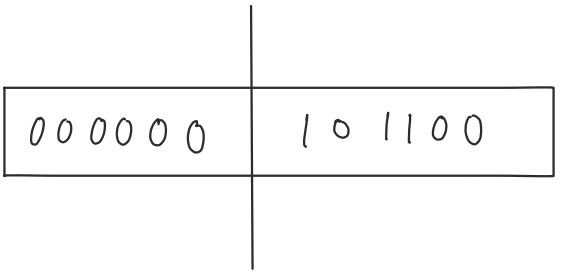


divisor 000011
dividend 101100

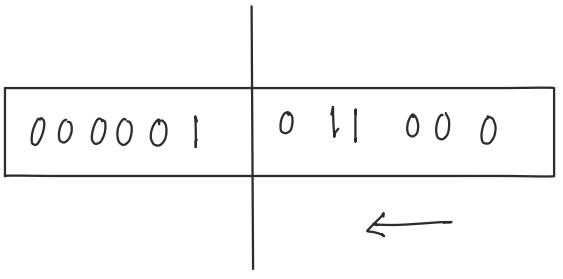
(1)



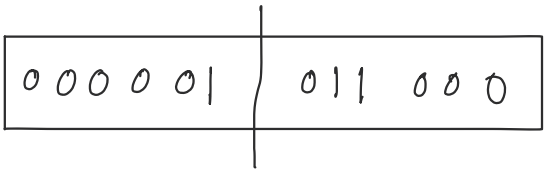
After sub

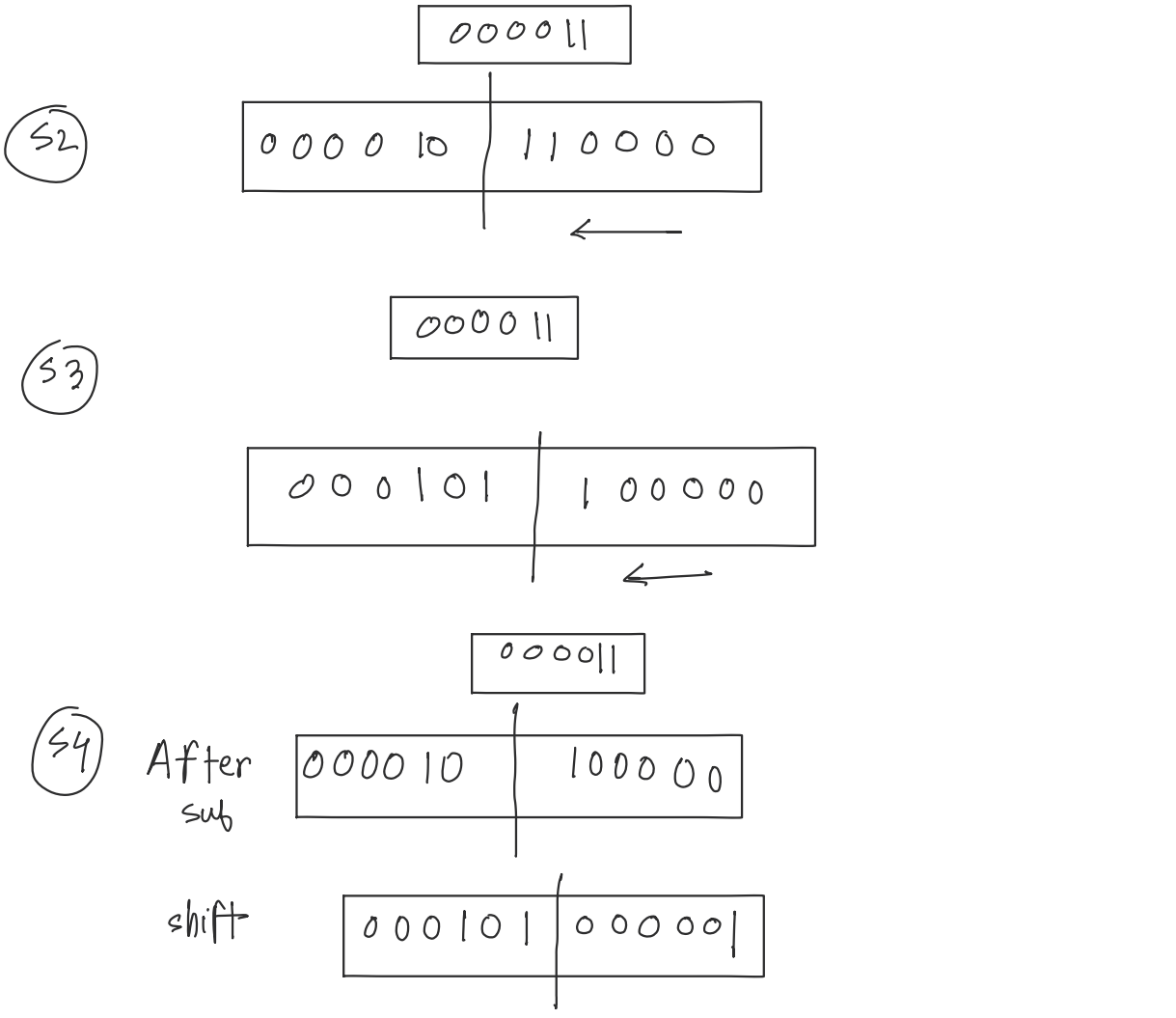


After shift



ultimately




000011
S2
0000 10 | 110000
000011
S3
000101 | 100000
000011
S4
After sub
000010 | 100000
shift
000101 | 000000
S5
000 010 | 000000
after sub
000100 | 000011
sub : 000001 0000 11
S6
000 010 | 000111
after shift
S7
000100 | 00 111 0
remainder portion 는 extra right shift 시켰기 때문
∴ remainder 000010

∴ last step এ একটা extra কাজ করা লাগবে। right shift. শুধু একটা register এ shift করছি। High, low থাকবে (register এ) so এটা করা possible.

xm এ 8 bit এর জন্য থাকবে।
full steps দেখানো লাগবে। (after sub, after shift *)
show the calculation বলা না থাকলেও সব লেখা লাগবে।

multiplication এর addition আলাদা করে দেখানো লাগবে না।
divisor এ subtraction.