

# California State Polytechnic University, Pomona

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

**COMPUTER ARCHITECTURE**

**ECE 4300**

CLASS ASSIGNMENT

# SRT Division

Prepared by

Maria N. Aponte

Presented to Professor

**Mohammed Aly**

October 21, 2025

Moore’s Law applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many adders to speed up multiply, but we

cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per step. The *SRT division* technique tries to predictseveral quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong predictions. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division, there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step. The accuracy of this fast method depends on having proper values in the lookup table.

SRT division is a hardware algorithm used to perform high-speed division in CPUs. It is named after its inventors, Sweeney, Robertson, and Tocher.

SRT tries to predict several bits of the quotient per step instead of just one (like in basic restoring or non-restoring division).

It uses a lookup table based o n a few bits of the dividend (or partial remainder), and the divisor. Then it used that prediction to decide what multiple of the divisor to substract next.

**Basic Division Reminder**

Let’s recall how **binary long division** works (1 bit at a time):

Example:  
Divide **13 (1101₂)** by **3 (11₂)**

Quotient = 0100 (which is 4)

Remainder = 1

Each step computes one quotient bit — this is slow.

**How SRT Speeds This Up**

SRT can compute more than one quotient bit per step, often 2 or 4 bits.

It does this by:

1. **Estimating** (predicting) the next quotient digit using a small lookup table.
2. **Subtracting or adding** a *multiple* of the divisor accordingly.
3. **Correcting** the prediction in later steps if it was slightly off.

So, instead of dividing bit-by-bit, it might divide nibble-by-nibble (4 bits).

**Example: Simple SRT (Radix-2) Example**

Using Radix-2 SRT (one bit at a time) to illustrate the process.  
Later using Radix-4 (2 bits per step).

Divide 13 (1101₂) by 3 (11₂) again.

Step 1: Setup

* Dividend = 1101
* Divisor = 11
* Normalize divisor ≈ 1.x form (conceptually)
* Partial remainder starts as 0.

We’ll perform 4 iterations (since dividend has 4 bits).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Step** | **Partial Remainger** | **Bring Down Bit** | **Subtract or Add (Divisor xq)** | **Quotient Bit** |
| 1 | 0 | 1 → PR = 1 | Subtract 1.1? PR still positive → = 1 | 1 |
| 2 | PR=remainder | Bring next bit down | Compute again | - |

This part looks similar to long division. But in SRT, we use a digit set of {-1, 0, +1} for quotient digits.

**Example: SRT Radix-4 Division (2 bits per step)**

Let’s do a small numeric example with decimal numbers to show how prediction works.

Devide:

Assume Radix-4 → each step predicts 2 bits (values: -2, -1, 0, 1, 2).

**Step 1: Normalize**

Scale so divisor ≈ 1.x  
For simplicity, just conceptually divide by 3.

**Step 2: Use Lookup Table**

Look at the most significant bits of remainder and divisor to guess quotient digit.

Let’s say we use 6 bits of remainder and 4 bits of divisor to index a small table that tells us:

“Based on these bits, subtract 1×D, 2×D, or add something.”

**Step 3: Iterations**

**Iteration 1:**

* Remainder = 25
* Predicted quotient digit (from lookup): q₁ = +2 (since 25/3 ≈ 8, and we are working in radix-4 → first digit = 2)
* Subtract: 2 × 3 = 6
* New remainder = 25 – 6 = 19

**Iteration 2:**

* Next step, remainder = 19
* Predict q₂ = +2 again
* Subtract 6 → New remainder = 13

**Iteration 3:**

* Remainder = 13
* Predict q₃ = +2 again
* Subtract 6 → New remainder = 7

**Iteration 4:**

* Remainder = 7
* Predict q₄ = +2 again
* Subtract 6 → New remainder = 1

Now combine quotient digits (each is base 4):  
Quotient ≈ 2222₄ = 2×4³ + 2×4² + 2×4 + 2 = 13634₁₀ (this is rough because we didn’t normalize properly — real hardware normalizes before this).

But the idea is clear: each step predicts multiple bits, subtracts a multiple of the divisor, and corrects later.

**Why It Works (and Why It’s Tricky)**

* In binary division, quotient digit ∈ {0, 1}.
* In SRT division, quotient digit ∈ {−2, −1, 0, 1, 2} (for radix 4).  
  This flexibility lets the hardware tolerate small errors in prediction, since later steps can correct them.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Basic Binary Division** | **SRT Division** |
| Quotient bits per step | 1 | 2-4 |
| Quotient digit values | 0, 1 | –2, -1, 0, 1, 2 |
| Uses loopup table? | no | yes |
| Can correct errors? | no | yes |
| Speed | Slower | Faster |
| Exmaple hardware | Simple ALU | Modern CPU devider unit |

**Real-World Note**

Modern CPUs like Intel and ARM use Radix-4 or Radix-8 SRT division:

* They predict 2 or 3 bits per iteration.
* They use a table lookup based on leading bits of the remainder and divisor.
* Later steps refine the quotient and remainder.

**Example:**

Let’s divide:

Perform Radix-4 SRT division.

Normalize the divisor so that .  
In binary, that means shifting it so that its MSB = 1.

Normalized values:

We’ll ignore the scaling exponent and just work on the mantissas (1.101 ÷ 1.1).

**Table Lookup Idea**

At each step, the hardware looks at:

* The most significant bits of the remainder (R), and
* The most significant bits of the divisor (D)

to choose .

For example, in a simplified radix-4 SRT:

|  |  |
| --- | --- |
| **Range of R/D** | **qᵢ (predicted quotient digit)** |
| ≥ +1.5 | +2 |
| ≥ +0.5 | +1 |
| ≥ −0.5 | 0 |
| ≥ −1.5 | -1 |
| < −1.5 | -2 |

(Values come from the lookup table stored in hardware.)

**Step-by-Step Iteration**

We will perform 2 iterations, since radix-4 generates 2 bits per iteration and our dividend is 4 bits long.

**Initial conditions:**

**Iteration 1**

Compute approximate ratio:

→ lookup table says q₁ = +1 (since ratio ≈ +1.5).

Now compute next partial remainder:

Why “×4”? Because radix-4 → we shift remainder by 2 bits (like multiplying by 4).

**Iteration 2**

Now:

→ lookup table → q₂ = +1 again.

Compute new remainder:

Since R₂ is negative, next steps (if we continued) would correct by choosing or 0 in later cycles.

**Collecting Quotient Digits**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iteration** | |  | | --- | |  |  |  | | --- | | **qᵢ** | | **Binary (2 bits)** |
| 1 | +1 | 01 |
| 2 | +1 | 01 |

Combine (radix-4 → each represents 2 bits):

Final Quotient ≈ 4–5 (close to 13 ÷ 3 = 4.33)  
Remainder small and negative (−1), so hardware corrects it to the exact final value later.

|  |  |
| --- | --- |
| **Concept** | **Meaning** |
| Radix-4 | 2 quotient bits per iteration |
| Digit set | {−2, −1, 0, +1, +2} allows small prediction errors |
| Lookup table | Uses top bits of remainder and divisor |
| Correction | Later iterations fix small remainder sign errors |
| Speed | About twice as fas as simple bit-by-bit division |

**Real Hardware Analogy**

Modern CPUs (e.g. Intel Core, ARM Cortex) use:

* Radix-4 or Radix-8 SRT (2 or 3 bits per step)
* Lookup tables for quotient prediction
* Carry-save adders to compute remainders fast
* Correction logic if prediction overshoots

That’s why SRT is standard in floating-point division units.

A diagram of a system

AI-generated content may be incorrect.