GCD-V1/V2

Euclidean Algorithm

- **Euclidean Algorithm** (or Euclid's algorithm)→ it was first described by the ancient Greek mathematician **Euclid of Alexandria** around **300 BCE** in his famous book *Elements* (Book VII).
- In Elements, he presented a simple step-by-step method to find the **greatest common divisor (GCD) of two integers.**
- This procedure — repeatedly subtracting or dividing until the remainder is zero — is essentially the same algorithm we still use today.

Euclid's GCD Algorithm

Inputs: external values `a`, `b`

Output: `gcd(a,b)`

```
procedure GCD(a, b):
    A ← a
    B ← b

    while (A ≠ B) do
        if (A > B) then
            A ← A - B
        else
            B ← B - A
        end if
    end while

    return A   // or return B, both are equal here
end procedure
```

Example

```
Start: a = 12, b = 42

b > a → b = b - a = 42 - 12 = 30
a = 12, b = 30

b > a → b = b - a = 30 - 12 = 18
a = 12, b = 18

b > a → b = b - a = 18 - 12 = 6
a = 12, b = 6

a > b → a = a - b = 12 - 6 = 6
a = 6, b = 6

Now a = b = 6 → ✅ GCD = 6
```

```
a=27, b=90
b > a → b = 90 - 27 = 63
a=27, b=63

b > a → b = 63 - 27 = 36
a=27, b=36

b > a → b = 36 - 27 = 9
a=27, b=9

a > b → a = 27 - 9 = 18
a=18, b=9

a > b → a = 18 - 9 = 9
a=9, b=9
```

✅ GCD = 9

Example: $a = 270, b = 192$

✅ GCD = 6

```
gcd(270, 192)
270 ÷ 192 = 1 remainder 78   → gcd(192, 78)
192 ÷ 78  = 2 remainder 36   → gcd(78, 36)
78 ÷ 36   = 2 remainder 6    → gcd(36, 6)
36 ÷ 6    = 6 remainder 0    → gcd(6, 0)
```

Examples

- $\gcd(12, 0) = 12$
- $\gcd(0, 42) = 42$
- $\gcd(0, 0) = \text{undefined (special case)}$

The **subtraction-based Euclid method** is most useful when:

- You want **minimal hardware (area-efficient)**.
- You're working with **small bit-widths** (like 8-bit operands).

Design a synchronous (clocked) Verilog module that computes the **greatest common divisor (GCD)** of two **8-bit unsigned** inputs `a` and `b` using Euclid's algorithm. The module exposes a simple handshake (`go / idle / busy / done`) and an `invalid` flag for the edge case `(a=0, b=0)`.

Interfaces

Inputs

- `clk` : system clock (rising-edge triggered).
- `rst` : synchronous reset, active-high.
- `go` : one-cycle start strobe, sampled only in `IDLE`.
- `a[7:0]`, `b[7:0]` : external **unsigned** operands.

Outputs

- `idle` : `1` only in `IDLE` (ready to accept a new job).
- `busy` : `1` during computation.
- `done` : **one-cycle pulse** when result is produced.
- `gcd[7:0]` : latched final result; remains stable in `DONE` and `IDLE`.
- `invalid` : `1` **only in** `IDLE` when `(a==0 && b==0)` (undefined GCD).

Functional Behavior

1. **Reset**
   - On `rst=1`, the FSM enters **IDLE**.
   - `idle=1`, `busy=0`, `done=0`, `invalid` reflects `(a==0 && b==0)`.
   - `gcd` cleared to 0.
2. **IDLE state**
   - `idle=1`. Inputs `a` / `b` are monitored **only** for validity/invalid indication.
   - If `(a==0 && b==0)`: `invalid=1`, `go` must be **ignored**.
   - If at least one input is non-zero: `invalid=0`.
     - A **one-cycle** `go=1` latches the current `a` and `b` into internal registers and transitions to **COMPUTE** on the next clock.
   - Any `go` when `invalid=1` does **not** start the operation.

3. **COMPUTE state**
    - `busy=1` (and `idle=0`, `done=0`).
    - Run Euclid's algorithm (subtractive or modulo; subtractive is acceptable) on the **latched** operands until termination:
        - If either internal operand becomes zero, the other holds the GCD.
        - (Equivalent termination if the two operands become equal; that value is the GCD.)
    - External changes on `a` / `b` during COMPUTE are **ignored**.
4. **DONE state**
    - On completion, assert `done=1` for **exactly one clock**; latch `gcd`.
    - After the `done` pulse, return to **IDLE**: `idle=1`, `busy=0`, `invalid` reflects current external `(a,b)` again.
    - `gcd` remains held until the next successful start or reset.

Minimal functional pins (8-bit)

Inputs (19):

- `clk` (1)
- `rst` (1)
- `go` (1)
- `a[7:0]` (8)
- `b[7:0]` (8)

Outputs (12):

- `gcd[7:0]` (8)
- `idle` (1)
- `busy` (1)
- `done` (1)
- `invalid` (1)

Power (2):

- `VDD` (1)
- `VSS/GND` (1)

👉 Minimum total = 19 + 12 + 2 = 33 pins.

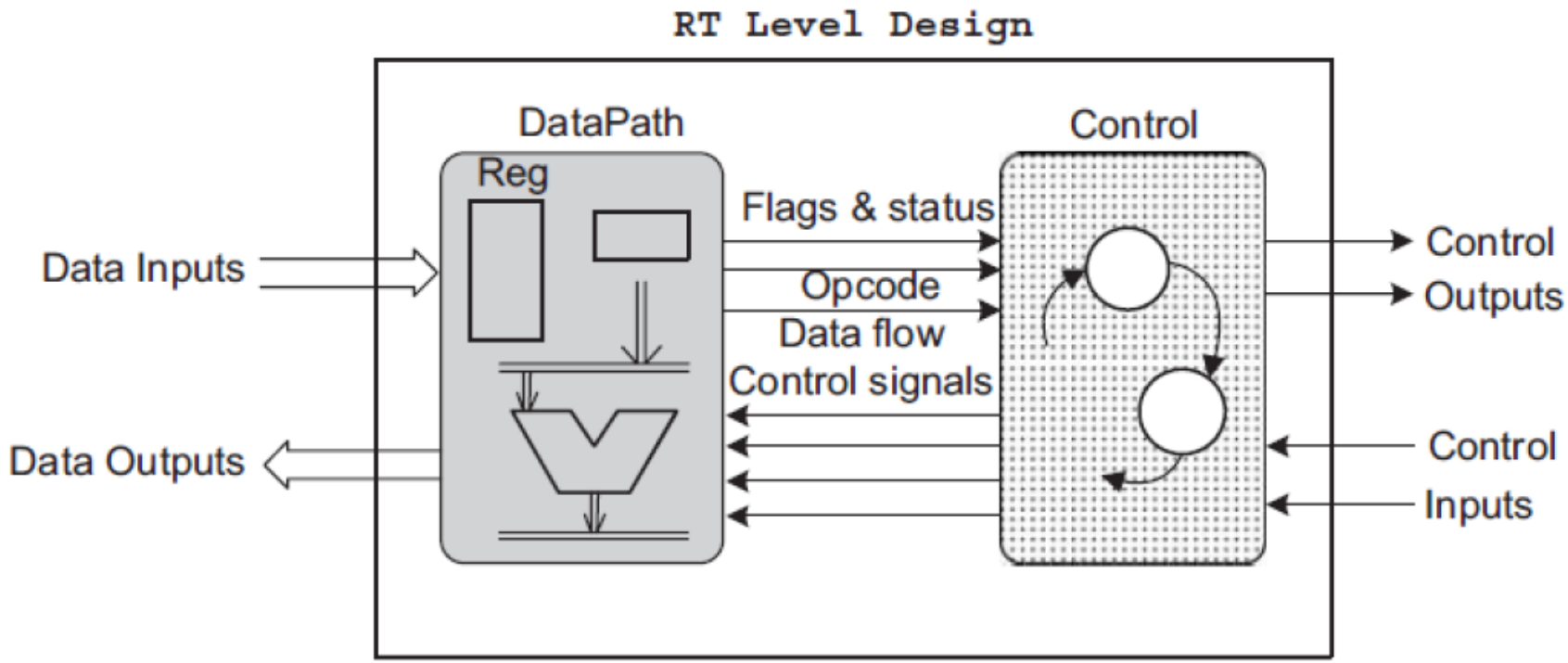
```
while (a != b) begin
    if (a > b) a = a - b;
    else       b = b - a;
end
```

- But in **synthesis (hardware)**:
  - The synthesizer must map your code to a fixed hardware structure.
  - A `while` loop whose number of iterations depends on *runtime data* (`a` and `b` values) has **no compile-time bound**.
  - Hardware cannot instantiate "a variable number of subtractors" depending on the data — so the loop is **not synthesizable**.

- Here, subtraction happens **over multiple clock cycles** until convergence.
- That's how you implement the "while loop" in hardware.

Controller/datapath partitioning

The first step in an RT level design is the partitioning of the design into a data part and a control part. The data part consists of data components and the bussing structure of the design and the control part is usually a state machine generating control signals that control the flow of data in the data part.



GCD-V2

Password-Protected Euclid GCD — Problem Statement

Purpose

Design a synchronous Verilog module that computes the **GCD of two 8-bit unsigned inputs** `a` and `b`, but only after the correct **password sequence** `1101` is entered.

The password is fed serially on a single input line `pw_bit`, sampled on each clock cycle while the system is i the `IDLE` state.

If the 4-bit password is correct → the system unlocks (`gcd_on=1`).
If the 4-bit password is wrong → the system stays locked (`lock=1`).

Once unlocked, the user may request a GCD computation. The system then runs the Euclidean subtraction algorithm, asserts status flags (`idle` , `busy` , `done` , `invalid_err`), and finally re-locks.

Interfaces

Inputs

- `clk` : system clock (rising-edge).
- `rst` : synchronous reset (active-high).
- `pw_bit` : serial password input bit, sampled every clock in IDLE.
- `go` : one-cycle start request (works only if in `GCD_START`, unlocked, and input valid).
- `a[7:0], b[7:0]` : unsigned operands.

Outputs

- `idle` : `1` in `IDLE` (waiting for password).
- `lock` : `1` in `IDLE` before correct password is entered.
- `gcd_on` : `1` while in `GCD_START`, `GCD_COMP`, **or** `DONE` (cleared upon return to `IDLE`).
- `busy` : `1` while in GCD computation state.
- `done` : pulses high for one clock cycle when GCD is ready.
- `invalid_err` : pulses high if `go=1` in `GCD_START` but inputs are invalid (`a=0` and `b=0`).
- `gcd[7:0]` : result register, holds computed GCD value.