CSE140: Components and Design Techniques for Digital Systems

High-Level Finite State Machines

Tajana Simunic Rosing
High-Level State Machine

- Some behaviors may be too complex to describe by using classical FSMs
- Soda dispenser
  - c: bit input, 1 when coin deposited
  - a: 8-bit input: value of the deposited coin
  - s: 8-bit input: cost of a soda
  - d: bit output, processor sets it to 1 when total value of deposited coins equals or exceeds cost of a soda
Which of the following makes the FSM design of this problem difficult?

A. 8-bit input/output
B. Tracking the current total
C. Multibit comparison
D. All of the above
E. None of the above
Benefits of HLSMs

• High-level state machine (HLSM) extends FSM with:
  – Multi-bit input/output
  – Local storage
  – Arithmetic operations

• Conventions
  – Numbers:
    • Single-bit: '0' (single quotes)
    • Integer: 0 (no quotes)
    • Multi-bit: “0000” (double quotes)
  – == for comparison equal
  – Multi-bit outputs must be registered via local storage
  – // precedes a comment
FSMs vs. HLSMs

How does the HLSM differ from the FSM for this problem?

A. The HLSM stores multibit data, but the FSM doesn’t ✓

B. The FSM stores the state but the HLSM doesn’t ✗

C. Implementing HLSM and FSM requires multibit data registers ✗

D. All of the above ✗

E. None of the above ✗
Which of the following are common between HLSMs and FSMs?

A. Transitions happen at the edge of a clock  ✔
B. They both have external complex data  ❌
C. All of the above  ❌
D. None of the above  ❌
RTL Design Process

• Capture the behavior
• Convert it to a circuit
  – High-level architecture (datapath and control path)
  – Datapath capable of HLSM's data operations
  – Design controller to control the datapath
Step 2: Create Datapath for Soda Dispenser

- Need \( \text{tot} \) register to keep track of the money deposited so far.
- Need 8-bit comparator to compare \( s \) (current sum) and \( a \) (target cost).
- Need 8-bit adder to update: \( \text{tot} = \text{tot} + a \).
- Connect everything.
- Create control IO.

Diagam with labels:

- Inputs: \( c \) (bit), \( a \) (8 bits), \( s \) (8 bits).
- Outputs: \( d \) (bit).
- Local registers: \( \text{tot} \) (8 bits).

Diagram specifics:

- \( \text{tot}_\text{ld} \), \( \text{tot}_\text{clr} \), \( \text{tot}_\text{lt}_s \).
- 8-bit adder.
- \( \text{Datapath} \).

Sources: TSR, Katz, Boriello, Vahid, Perkowski.
According to the current design, under which of the following conditions does the register output ‘tot’ change at the rising clock edge?

A. Whenever the value of the coin inserted (‘a’) changes
B. Whenever the cost of the soda (‘s’) changes
C. When the signal tot_ld becomes high
D. When the signal tot_clr becomes high
E. Both C. & D.
Step 3: Connect Datapath to a Controller

- Controller’s inputs
  - External input $c$ (coin detected)
  - Input from datapath comparator’s output, which we named $tot_{lt_s}$
- Controller’s outputs
  - External output $d$ (dispense soda)
  - Outputs to datapath to load and clear the $tot$ register

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Step 4 – Derive the Controller’s FSM

- FSM has the same states and arcs as HLSM
- Replace all references to the data elements in the HLSM with appropriate control signals & values

Inputs: c, tot_lt_s (bit)
Outputs: d, tot_ld, totClr (bit)

Controller

- Init: d=0, tot_clr=1
- Wait: c
- Add: tot_ld=1
- Disp: c'* tot_lt_s

Datapath

- 8-bit <
- 8-bit adder

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Step 4: Finalize the Implementation

Implement the FSM as a state register and logic

Inputs: $c, \text{tot}_{\text{lt}}_s$ (bit)
Outputs: $d, \text{tot}_{\text{ld}}, \text{tot}_{\text{clr}}$ (bit)

Controller

Init

Wait

Add

Disp

Init

Wait

Add

Disp

\[ d = 0 \quad \text{tot}_{\text{clr}} = 1 \]

\[ c' \ast \text{tot}_{\text{lt}}_s \]

\[ d = 1 \]

\[ \text{tot}_{\text{ld}} = 1 \]

\[ \text{tot}_{\text{clr}} \]

\[ \text{tot}_{\text{ld}} \]

\[ s1 \quad s0 \quad c \quad \text{tot}_{\text{lt}}_s \]

\[ n1 \quad n0 \quad d \quad \text{tot}_{\text{ld}} \quad \text{tot}_{\text{clr}} \]

0 0 0 0 0 1 0 0 1 0 1
0 0 0 1 0 1 0 0 1 1
0 0 1 0 0 1 0 0 1 1
0 0 1 1 1 0 1 0 0 0
0 1 0 0 1 1 0 0 1 0
0 1 0 1 0 1 0 0 0 0
0 1 1 0 0 1 0 0 0 0
0 1 1 1 1 1 0 0 0 0
1 0 0 0 0 0 1 0 0 0
1 0 0 1 0 0 1 0 0 0
1 1 0 0 0 1 0 0 0 0

5 k-maps

Sources: TSR, Katz, Boriello, Vahid, Perkowski
RTL Example: Isotope Tracker

• A chemistry lab in UCSD has approached CSE140 students to design a circuit which determines the amount of isotopes currently remaining in an experimental chamber

The circuit receives three inputs, ‘R,’ ‘I,’ and ‘H’:
- I: tells when new isotopes are added
- R: a 16 bit number representing the quantity of new isotopes injected
- H: tell when isotopes are halved

Output
- The amount of isotopes remaining
Another RTL Design:
Laser-Based Distance Measurer

- Laser-based distance measurement – pulse laser, measure time \( T \) to sense reflection
  - Laser light travels at speed of light, \( 3 \times 10^8 \) m/sec
  - Distance is thus \( D = T \) sec * \( 3 \times 10^8 \) m/sec / 2
Laser-Based Distance Measurer I/O

- Inputs/outputs
  - $B$: bit input, from button, to begin measurement
  - $L$: bit output, activates laser
  - $S$: bit input, senses laser reflection
  - $D$: 16-bit output, to display computed distance
Laser-Based Distance Measurer: HLSM

- Declare inputs, outputs, and local storage
  - Dreg required for multi-bit output
- Create initial state, name it S0
  - Initialize laser to off (L:='0')
  - Initialize displayed distance to 0 (Dreg:=0)

\[ L := '0' \quad \text{// laser off} \]
\[ \text{Dreg} := 0 \quad \text{// distance is 0} \]
Laser-Based Distance Measurer: HLSM

• Add another state, **S1**, that waits for a button press
  – **B'** – stay in **S1**, keep waiting
  – **B** – go to a new state **S2**
Laser-Based Distance Measurer: HLSM

- Add a state $S_2$ that turns on the laser ($L:='1'$)
- Then turn off laser ($L:='0'$) in a state $S_3$
Laser-Based Distance Measurer: HLSM

- Stay in S3 until sense reflection (S)
- To measure time, count cycles while in S3
  - To count, declare local storage Dctr
  - Initialize Dctr to 0 in S1. In S2 would have been O.K. too.
    - Don't forget to initialize local storage—common mistake
  - Increment Dctr each cycle in S3
Laser-Based Distance Measurer: HLSM

- Once reflection detected (S), go to new state S4
  - Calculate distance
  - Assuming clock frequency is $3 \times 10^8$, $Dctr$ holds number of meters, so $Dreg := Dctr/2$
- After S4, go back to S1 to wait for button again

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Laser-Based Distance Measurer: Create a Datapath

- HLSM data I/O → DP I/O
- HLSM local storage → reg
- HLSM state action and transition condition data computation → Datapath components and connections

DistanceMeasurer

**Inputs:** B (bit), S (bit)
**Outputs:** L (bit), D (16 bits)
**Local storage:** Dreg, Dctr (16 bits)

---

**Datapath**

- **Add1**: add(16)
- **Shr1**: shiftR1(16)
- **Dreg_clr**
- **Dreg_ld**
- **Dctr_clr**
- **Dctr(ld**

---

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Laser-Based Distance Measure: Connecting the Datapath to a Controller

Diagram:
- Controller
- Datapath
- 300 MHz Clock
- L to laser from sensor
- S to display from button
- D to display
- 16

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Laser-Based Distance Measurer: Derive the Controller FSM

- FSM has same states, transitions, and control I/O
- Achieve each HLSM data operation using datapath control signals in FSM

**DistanceMeasurer**

- **Inputs:** B (bit), S (bit)
- **Outputs:** L (bit), D (16 bits)
- **Local storage:** Dreg, Dctr (16 bits)

**Controller**

- **Inputs:** B, S
- **Outputs:** L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_ld

**Datapath**

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Laser-Based Distance Measurer: Simplify the Controller FSM

- Same FSM, using convention of unassigned outputs implicitly assigned 0

Some assignments to 0 still shown, due to their importance in understanding desired controller behavior
# RTL Design Process

<table>
<thead>
<tr>
<th>Step 1: Capture behavior</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Capture a high-level state machine</em></td>
<td>Describe the system’s desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is “high-level” because the transition conditions and the state actions are more than just Boolean operations on single-bit inputs and outputs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2: Convert to circuit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A <em>Create a datapath</em></td>
<td>Create a datapath to carry out the data operations of the high-level state machine.</td>
</tr>
<tr>
<td>2B <em>Connect the datapath to a controller</em></td>
<td>Connect the datapath to a controller block. Connect external control inputs and outputs to the controller block.</td>
</tr>
<tr>
<td>2C <em>Derive the controller’s FSM</em></td>
<td>Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.</td>
</tr>
</tbody>
</table>
Remember the Soda HLSM?

Inputs: c (bit), a(8 bits), s (8 bits)
Outputs: d (bit)
Local registers: tot (8 bits)
RTL Delay: A Circuit May Have Numerous Paths

- Frequency is limited by *longest register-to-register delay* – *critical path*
- Paths can exist
  - In the datapath
  - In the controller
  - Between the controller and datapath
  - May be hundreds or thousands of paths

\[ t_{ccq} + t_{cd} > t_{hold} + t_{skew} \]
\[ T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew} \]
### RTL Design Process: Datapath components

#### Datapath Components

- **Adder**
  - Input: A, B
  - Output: S
  - Equation: \( S = A + B \)
  - Conditions:
    - \( \text{clk} \uparrow \text{and} \ clr=1: \ Q=0 \)
    - \( \text{clk} \uparrow \text{and} \ ld=1: \ Q=I \)
    - Else: \( Q \) stays same

- **Comparator**
  - Input: A, B
  - Output: Q
  - Conditions:
    - \( A < B: \ lt=1 \)
    - \( A = B: \ eq=1 \)
    - \( A > B: \ gt=1 \)

- **Shift Register**
  - Input: I
  - Output: Q
  - Conditions:
    - \( s0=0: \ Q=I0 \)
    - \( s0=1: \ Q=I1 \)

- **Subtractor**
  - Input: A, B
  - Output: S
  - Equation: \( S = A - B \)
  - Conditions:
    - \( \text{clk} \uparrow \text{and} \ clr=1: \ Q=0 \)
    - \( \text{clk} \uparrow \text{and} \ inc=1: \ Q=Q+1 \)
    - Else: \( Q \) stays same

- **Multiplier**
  - Input: A, B
  - Output: P
  - Equation: \( P = A \times B \)
  - Conditions:
    - \( \text{clk} \uparrow \text{and} \ W_e=1: \ RF[W_a]= W_d \)
    - \( R_e=1: \ R_d = RF[R_a] \)

Sources: TSR, Katz, Boriello, Vahid, Perkowski
More Datapath Examples

(a) \[ P_{\text{reg}} = X + Y + Z \]

(b) \[ P_{\text{reg}} = P_{\text{reg}} + X \]

(c) \[ P_{\text{reg}} = X + Y; \quad \text{reg}Q = Y + Z \]

(d) \[ k=0: P_{\text{reg}} = Y + Z \]
\[ k=1: P_{\text{reg}} = X + Y \]
HLSM Array Example

ArrayEx

Inputs: (none)
Outputs: P (11 bits)
Local storage: A[4](11 bits)
Preg (11 bits)

Preg := 0
A[0] := 9

A[0] == 8)

A[0] == 8


(b)

ArrayEx

Inputs: A_eq_8
Outputs: A_s, A_Wa0, ...

Preg_clr = 1
A_s = 0
A_Wa1 = 0, A_Wa1 = 0
A_We = 1

A_eq_8)

A_s = 1
A_Wa1 = 0, A_Wa0 = 1
A_We = 1
A_Ra1 = 0, A_Ra0 = 0
A_Re = 1

Out1

Preg_Id = 1

Controller

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Video is a series of frames (e.g., 30 per second)
Most frames similar to previous frame
  Compression idea: just send difference from previous frame
Video Compression – Sum of Absolute Differences

- If two frames are similar just send a difference instead
  - Compare corresponding 16x16 “blocks”
    - Treat 16x16 block as 256-byte array
  - Compute the absolute value of the difference of each array item
  - Sum the differences
    - If above a threshold, send a complete frame for second frame
    - Else send the difference

Each is a pixel, assume represented as 1 byte (actually, a color picture might have 3 bytes per pixel, for intensity of red, green, and blue components of pixel)
Sum-of-Absolute Differences: High-level FSM

- **S0**: wait for go
- **S1**: initialize sum and index
- **S2**: check if done (i>=256)
- **S3**: add difference to sum, increment index
- **S4**: done, write to output sad_reg

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)
Sum-of-Absolute Differences: Datapath

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)

S0

S1

S2

S3

S4

\( !go \)

\( go \)

\( \text{sum} = 0 \)

\( \text{i} = 0 \)

\( \text{i} < 256 \)

\( \text{sum} = \text{sum} + \text{abs}(A[i] - B[i]) \)

\( \text{i} = \text{i} + 1 \)

\( !(! \text{i} < 256) (\text{i}\_\text{lt}\_256) \)

\( \text{sad\_reg} = \text{sum} \)

Datapath
Sum-of-Absolute Differences: Connect datapath and controller & specify the controller FSM

\[ \text{sum} = 0 \quad \text{sum}_{-} \text{clr} = 1 \]
\[ i = 0 \quad i_{-} \text{clr} = 1 \]
\[ i < 256 \quad i_{-} \text{lt}_{-} 256 \]
\[ \text{sum} = \text{sum} + \text{abs}(A[i] - B[i]) \]
\[ \text{sum}_{-} \text{ld} = 1; \ AB_{-} \text{rd} = 1 \]
\[ i = i + 1 \quad i_{-} \text{inc} = 1 \]
\[ \text{sad}_{-} \text{reg} = \text{sum} \]
\[ \text{sad}_{-} \text{reg}_{-} \text{ld} = 1 \]

Controller

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Earlier sum-of-absolute-differences example
- Started with high-level state machine
- C code is an even better starting point -- easier to understand

**C code**

```c
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint i;
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] - B[i]);
        i = i + 1;
    }
    return sum;
}
```

**Inputs**: A, B [256](8 bits); go (bit)
**Outputs**: sad (32 bits)
**Local storage**: sum, sadreg (32 bits); i (9 bits)
Converting from C to High-Level State Machine

- Convert each C construct to equivalent states and transitions
  - **Assignment** statement
    - Becomes one state with assignment
  - **If-then** statement
    - Becomes state with condition check, transitioning to “then” statements if condition true, otherwise to ending state
    - “then” statements would also be converted to states

```
target = expression;
```

```
if (cond) {
    // then stmts
}
```

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Converting from C to High-Level State Machine

- **If-then-else**
  - Becomes state with condition check, transitioning to “then” statements if condition true, or to “else” statements if condition false

- **While loop statement**
  - Becomes state with condition check, transitioning to while loop’s statements if true, then transitioning back to condition check
Converting from C to HLSM: Example

- Simple example: computing the maximum of two numbers
  - Convert if-then-else statement to states (b)
  - Then convert assignment statements to states (c)

Inputs: uint X, Y
Outputs: uint Max

```c
if (X > Y) {
    Max = X;
} else {
    Max = Y;
}
```

(a)
Example: SAD C code to HLSM

- Convert each construct to states
  - Simplify states
- Use RTL design process to convert to circuit
- However, only a subset of C can be easily converted
  - Can use language other than C

```c
Inputs: byte A[256], B[256]  
bit go;  
Output: int sad
main()
{
  uint sum; short uint i;
  while (1) {
    while (!go) {  
      sum = 0;  
      i = 0;  
      while (i < 256) {  
        sum = sum + abs(A[i] - B[i]);  
        i = i + 1;
      }
      sad = sum;
    }
  }
}
```
Circuit vs. Microprocessor

- **Circuit:**
  - Two states (S2 & S3) for each i, 256 i's → 512 clock cycles

- **Microprocessor:**
  - Loop (for i = 1 to 256), but for each i, must move memory to local registers, subtract, compute absolute value, add to sum, increment i
  - Each loop iteration is approx 6 cycles per array item → 256*6 = 1536 cycles

- **Circuit is ~3 times (300%) faster**
- It is possible to build a circuit that is much faster than this
  - think about how to leverage parallelism in HW!

\[
\begin{align*}
S2 & : \text{(i<256)} \\
S3 & : \text{i:=i+1, sum:=sum+abs(A[i]-B[i])}
\end{align*}
\]
Data vs. Control Dominated RTL Design

- Data dominant design: extensive datapath, simple controller
- Control dominant design: complex controller, simple datapath

Example: Filter
Converting digital input stream to a new digital output stream
- e.g.: remove noise
  - 180, 180, 181, 180, 240, 180, 181
  - 240 is probably noise, filter might replace by 181
- Simple filter: output average of the last $N$ values
  - Small $N$: less filtering
  - Large $N$: more filtering, but less sharp output
Data Dominated RTL Design Example: FIR Filter

- **FIR filter**
  - “Finite Impulse Response”
  - A configurable weighted sum of past input values
  - \[ y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-1) + c_2 \cdot x(t-2) \]
    - Above known as “3 tap”
    - Tens of taps more common
    - Very general filter – User sets the constants \((c_0, c_1, c_2)\) to define a specific filter

- **RTL design**
  - Step 1: Create HLSM
    - Very simple states/transitions

**Notation:**
- **Inputs:** \(X\) (12 bits)
- **Outputs:** \(Y\) (12 bits)
- **Local storage:** \(x_{t0}, x_{t1}, x_{t2}, c_0, c_1, c_2\) (12 bits)
- **Yreg** (12 bits)

**Circuit Diagram:**
- Digital filter

**RTL Design Steps:**
- **Init**
  - \(Y_{reg} := 0\)
  - \(x_{t0} := 0\)
  - \(x_{t1} := 0\)
  - \(x_{t2} := 0\)
  - \(c_0 := 3\)
  - \(c_1 := 2\)
  - \(c_2 := 2\)

- **FC**
  - \(Y_{reg} := c_0 \cdot x_{t0} + c_1 \cdot x_{t1} + c_2 \cdot x_{t2}\)
  - \(x_{t0} := X\)
  - \(x_{t1} := x_{t0}\)
  - \(x_{t2} := x_{t1}\)

**Notes:**
- Assumes constants set to 3, 2, and 2

Sources: TSR, Katz, Boriello, Vahid, Perkowski
FIR Filter: Create datapath

Step 2: Create datapath
- Create a chain of xt registers to hold past values of x
- Instantiate registers for c0, c1, c2
- Instantiate multipliers to compute c*x values
- Instantiate adders
- Add circuitry to allow loading of c register

\[ y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2) \]

Steps 3 & 4: Connect to controller, & create FSM:
No controller needed!
FIR Filter: Design the Circuit

- Create datapath
- Connect control and datapath
- Derive FSM
  - Set clr and ld lines appropriately

Datapath for 3-tap FIR filter

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Comparing the FIR circuit to a software implementation

- **Circuit**
  - Adder has 2-gate delay, multiplier has 20-gate delay
  - Longest past goes through one multiplier and two adders
    - $20 + 2 + 2 = 24$-gate delay
  - **100-tap filter has 34-gate delay**: 1 multiplier & 7 adders on longest path

- **Software**
  - 100-tap filter: 100 multiplications, 100 additions.
  - If 2 instructions per multiplication, 2 per addition. Say 10-gate delay per instruction.
  - $(100\times2 + 100\times2)\times10 = \textbf{4000 gate delays}$

$$y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2)$$
Another RTL Design Example: Bus Interface

- Master processor can read register from any peripheral
  - Each register has unique 4-bit address
  - Assume 1 register/peripheral
- Sets \( rd = 1, A = address \)
- Appropriate peripheral places register data on 32-bit \( D \) lines
  - Peripheral’s own address given on \( Faddr \) inputs
Bus Interface: Create a datapath

Inputs: rd (bit); Q (32 bits); A, Faddr (4 bits)
Outputs: D (32 bits)
Local register: Q1 (32 bits)

Steps for creating datapath:
1. Define its inputs/outputs
2. Instantiate registers
3. Instantiate components
4. Connect components
Bus Interface: Connect datapath to controller & derive controller’s FSM

**Datapath**

**Bus interface**

**Inputs:** rd, A_eq_Faddr (bit)

**Outputs:** Q1_Id, D_en (bit)

- **WaitMyAddress**
  - D_en = 0
  - Q1_Id = 1

- **SendData**
  - D_en = 1
  - Q1_Id = 0

- \( D = Z \)
- \( Q1 = Q \) \((A = Faddr) \text{ and } rd\)
- \( D = Q1 \) \((A = Faddr) \text{ and } rd'\)

- Inputs: rd, A_eq_Faddr (bit)
- Outputs: Q1_Id, D_en (bit)

- \( D_en = 0 \)
- \( Q1_Id = 1 \)

- \( D_en = 1 \)
- \( Q1_Id = 0 \)

---

Sources: TSR, Katz, Boriello, Vahid, Perkowski
RTL Summary

• Datapath and Control Design

• RTL Design Steps
  1. Define the high level state machine
  2. Create datapath
  3. Connect datapath with control
  4. Implement the FSM

• Timing analysis – critical path in more complex circuits
  – Watch out for all possible long paths (e.g. datapath to FSM, FSM control logic, datapath logic etc)
Common RTL Design Pitfall Involving Storage Updates

**Questions**
- Value of Q after state A?
- Final state is C or D?

**Answers**
- Q is NOT 99 after state A
- Q is 99 in state B, so final state is C
- Storage update actions in state occur *simultaneously* on *next* clock edge
  - Thus, order actions are written is irrelevant
  - A's actions same if:
    - Q:=R  R:=99   or
    - R:=99  Q:=R

Local storage: R, Q (8 bits)
Common RTL Design Pitfall Involving Storage Updates

- New HLSM using extra state so read of R occurs after write of R

Local storage: R, Q (8 bits)

- R := 99
- Q := R
- R := R + 1
- Q := R

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Simple data encryption/decryption device

- B = 1, set offset O = I[0:31]
- B = 0 e = 1: encrypt mode: output J = I + O
- B = 0 e = 0: decrypt mode: get I = J - O
Design from “C” code

Inputs: byte a, byte b, bit go
Outputs: byte gcd, bit done

GCD:
while(1) {
    while(!go);
    done = 0;
    while ( a != b ) {
        if( a > b ) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    gcd = a;
    done = 1;
}
Hot Water Detector

Create an alarm system that sets alarm=1 when the average temperature of four consecutive samples CT meets or exceeds a threshold WT. Signal clr=1 disables the alarm.
Fibonacci Lookup Table

- Design a lookup table 256 x 256 bit that stores Fibonacci #s:
  - \( F_n = 0 \) if \( n = 0 \), \( 1 \) if \( n = 1 \), \( F_{n-1} + F_{n-2} \) otherwise (\( n < 256 \))
Finish HLSM Design

• Design an 8-bit counter using RTL:
  – When input E = 1, it counts even numbers (0, 2, 4, 6, ..) and when E = 0, it counts odd numbers (1, 3, 5, 7, ..).
  – When input CLR = 1 and E = 1, then it clears the output to 0; if CLR = 1 and E = 0, it sets output to “00000001”.
  – If you were initially counting even(odd) numbers, and E flips, then the output changes to the nearest greater odd (even) value.