# Chapter 6

## SEQUENTIAL CIRCUITS: *LARGE DESIGNS*



# In this chapter

- Recall from Ch1:
  - Complex sequential circuit = data path + control unit
- Data path types
- Control unit types
- Performance parameters
- Design examples
  - Unsigned sequential multiplier
  - 2's complement sequential multiplier
  - A simple GPU



# Complex Data Paths

- Include combinational circuit modules
  - ALU, MUXs, decoders, wired-logic, etc.
  - Sometimes custom combinational circuits
- Include small sequential modules
  - Registers and counters
  - Sometimes custom sequential circuits
- Can include buses
  - With buffers and tristate buffers
- Include wires for interconnecting the modules creating multiple paths for data
  - Each path identifies a unique data path operation
    - Computation, memory access, etc.



# Control Units

1. Monitor events as input signals
   - External event signals
     - E.g., an "start" signal that starts a task performed on the data path
   - Data path event signals
     - Arithmetic overflow flag signal
     - A counter reaching a target value
     - Etc.
2. Generate control as output signals
   - Signals to control data path modules
     - E.g., ALU function signals, MUXs' selection signals, register enable signals, etc.
   - Signals to other interfacing modules (if any)
     - E.g., A "done" signal when done completing a task



# Register Transfer Notation (RTN)

- Formally describes a data path operation
- May use an arbitrary or an HDL syntax
- Examples:
  - CNTR ← CNTR + 1 //incrementing counter
  - CNTR <= CNTR + 1; //Verilog HDL
  - R ← R[7]//R[7:1] //Arithmetic right shift
  - R <= R >>> 1; //arithmetic right shift (Verilog)
  - R <= {R[7], R[7:1]}; //arithmetic right shift (Verilog)
  - M[x] ← R; //memory transfer (write)
  - R ← M[x]; //memory transfer (read)
  - Etc.



# Data Path Types

1. Single-cycle
   - Performs an operation specified by one or more RTNs during a single clock cycle
   - Uses more hardware but simple controller
   - Can be the lowest (e.g., execute a program)
2. Multi-cycle
   - Performs an operation specified by one or more RTNs using multiple clock cycles
   - Uses less hardware but more complex controller
     - Operations share hardware modules
   - Faster than single-cycle
3. Pipelined
   - Operates like an assembly line
   - Efficient when performing stream of operations
     - Stream of instructions
     - Stream of FLOPs
     - Etc.
   - Multiple clock cycles per operation
   - Concurrent processing
   - More hardware like single-cycle
   - High performance

# Design Example (Comparing different data paths)

- Problem: Consider two complex RTNs:
  - R ← A + B + C + D and R ← A + B + C - D
- Each called fused for involving three or more operands
  - 3-operand fused instructions common in some modern processors
    - E.g., Intel's multiply-add instruction
      - R ← A + B * C
      - Usefull for matrix operations (e.g., matrix multiplication)
    - One instruction instead of two
      - FMA vs. FMUL and FADD
- Advantage of fused RTN:
  - Reduces arithmetic rounding errors (refer to FP arithmetic)
  - Reduces number of instructions
- Complex fused RTNs application
  - Consider reconfigurable CPU



# 1. Single-Cycle Data Path

1. One clock cycle to perform R ← A + B + C + D or R ← A + B + C - D
2. Hardware required
   - two adders
   - One adder/subtractor
   - One register
3. Controller required
   - A *mode* signal deciding A + B + C + D or A + B + C − D

- Estimated data path clock period = ?

$$\tau \geq 2\Delta_{ADD} + \Delta_{ADD/SUB} + \tau_{st} + \tau_{cq} + \tau_{cs}$$



# 2. Multi-Cycle Data Path

- One or more clock cycles per RTN
  - Divide complex RTNs into simple ones
  - Multiple clock cycles to perform R ← A + B + C + D or R ← A + B + C - D
- More complex controller
  - Operations are done in steps
- Less hardware
  - One adder/subtractor, two MUXs, and a register
  - Data path modules used multiple times
    - E.g., adder/subtrcator module used 3 times
- Estimated data path clock period = ?

Example

```
Cycle 1: R ← A
Cycle 2: R ← R + B
Cycle 3: R ← R + C
Cycle 4: if mode == 0 then
           R ← R + D
         else
           R ← R – D
```

$$\tau \geq \Delta_{MUX1} + \Delta_{ADD/SUB} + \Delta_{MUX2} + \tau_{st} + \tau_{cq} + \tau_{cs}$$ Smaller than $\tau_{single-cycle}$



# 3. Pipelined Data Path

- Three clock cycles per RTN
  - Cycle 1: $X = A + B$
  - Cycle 2: $Y = X + C$
  - Cycle 3: $R = Y \pm D$
- Hardware similar to single-cycle except organized into stages
- Efficient when performing $A_i + B_i + C_i \pm D_i$ for $i = 0$ to $N-1$ for large N
- Requires less total time to complete a job
- Estimated clock period = ?

$$\tau \geq \Delta_{ADD/SUB} + \tau_{st} + \tau_{cq} + \tau_{cs}$$ Smaller than $\tau_{single-cycle}$ and $\tau_{multi-cycle}$



# Charting Pipeline Behavior

Final Stage →

| | | | | | |
|---|---|---|---|---|---|
| Stage 3 | | | R0 ← Y0 ± D0 | R1 ← Y1 ± D1 | … |
| Stage 2 | | Y0 ← X0 + C0 | Y1 ← X1 + C1 | Y2 ← X2 + C2 | … |
| State 1 | X0 ← A0 + B0 | X1 ← A1 + B1 | X2 ← A2 + B2 | X3 ← A3 + B3 | … |
| Cycle | 1 | 2 | 3 | 4 | … |

Horizontal Organization

Final Stage ↓

| Cycle | Stage 1 | Stage 2 | Stage 3 |
|---|---|---|---|
| 1 | X0 ← A0 + B0 | | |
| 2 | X1 ← A1 + B1 | Y0 ← X0 + C0 | |
| 3 | X2 ← A2 + B2 | Y1 ← X1 + C1 | R0 ← Y0 ± D0 |
| 4 | X3 ← A3 + B3 | Y2 ← X2 + C2 | R1 ← Y1 ± D1 |
| 5 | X4 ← A4 + B4 | Y3 ← X3 + C3 | R2 ← Y2 ± D2 |
| … | … | … | … |

Vertical Organization



# Performance Parameters

- Processing time (T)
  - Time required to perform a job (e.g., *N* tasks)
- Speedup
  - Calculated as a ratio
    - $\frac{T_{slow}}{T_{fast}}$, T of slower system divided by T of faster system
  - E.g., System A is 1.5 times faster than system B
- Efficiency
  - Calculated as a ratio
    - $\frac{S}{S_{ideal}}$, speedup divided by ideal speedup
  - E.g., 100% efficiency when all hardware modules utilized all the time
- Throughput
  - Calculated as ratio
    - $\frac{N}{T}$, Number of tasks performed divided by T
  - E.g., number of tasks performed per second

# Speedup (Pipelining vs. Single-cycle)

- *N* tasks: $A_i + B_i + C_i \pm D_i$ for $i = 0$ to $N$-1

$$Speedup = \frac{T_{single-cycle}}{T_{pipeline}}$$

Assume $\tau_{\text{single-cycle}} \simeq K\,\tau_{\text{pipeline}}$

- For $N = 3$ and $k = 3$, speedup = ? 1.8
- For N = 1000, k = 3, speedup = ? 2.99
- For very large $N$? (e.g., as $N \to \infty$) Speedup approaches k, the number of stages



# Efficiency of Pipelining

- *N* tasks: $A_i + B_i + C_i \pm D_i$ for $i = 0$ to $N$-1

$$Efficiency = \frac{Speedup}{K}$$

- For $N = 3$ and $k = 3$, Efficiency = ? 60%
- For $N = 1000$ and $k = 3$, Efficiency = ? 99.8%
- For very large $N$? (e.g., as $N \to \infty$) 100%



# Throughput of Pipelining

- *N* tasks: $A_i + B_i + C_i \pm D_i$ for $i = 0$ to $N$-1

$$Throughput = \frac{N}{T_{pipeline}}$$

- For N = 3 and k = 3, the throughput is about? $0.6\,\tau^{-1}$
- For N = 1000 and k = 3, the throughput is about? $0.998\,\tau^{-1}$
- Throughput as $N \to \infty$? $\tau^{-1}$ or $f$
- Standards
  - MIPS
    - E.g., 100 MIPS CPU
  - FLOPS
    - E.g., 1T FLOPS system
  - Benchmarks, more typical
    - SPEC CPU2006 for measuring performance of computer systems
    - SPECviewperf for measuring performance of computer-graphic systems



# Types of Control Units

1. Modeled as FSD for multi-cycle data paths
2. Micro-programmed Control (programmable)
   - Easy to modify after implementation
   - Also used if FSD would be very large
3. Pipeline control for pipelined data paths



# 1. FSD-based Control Unit

- Design steps:

1. Draw FSD for the controller
   - Assume CU triggered by external signal *start*
   - 4 clock cycles to perform $A + B + C \pm D$
   - Specify data path operations with RTNs
2. Complete design (assuming all structural)
   1. Determine specific data path control signals
   2. Draw detailed block diagram the controller
   3. Construct truth tables
   4. Find minimal expressions
   5. Combine with data path to complete design

```
Cycle 1: R ← A
Cycle 2: R ← R + B
Cycle 3: R ← R + C
Cycle 4: if mode == 0 then
             R ← R + D
         else
             R ← R – D
```



# 2. Microprogrammed Control

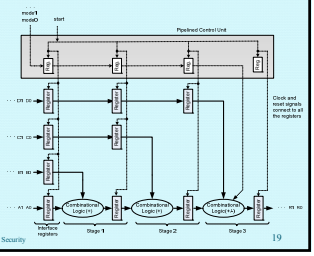
- Described as a program
  - Conditional statements use signal values
  - RTNs specifies data path operations
  - Branching changes program flow
- Tree pieces of hardware:
  - A 2-function counter, called microprogram counter (MPC)
  - A memory called control memory (CM) stores micro-instructions representation in binary
  - A MUX controls the functions of MPC

Example

```
0:if start == 0 then go to 0;
1: R ← A;
2: R ← R + B;
3: R ← R + C;
4: if mode == 0 then  R ← R + D,  go to 0;
5: R ← R - D,  go to 0;
```

# 3. Pipeline Control

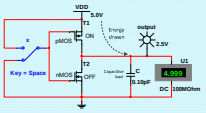
- As oppose to other two controllers, signals are generated at same time but applied to data path at different times
- Example, signal *mode* enters pipeline but not used until stage 3






# Circuit Energy Consumption

- Energy is a function of capacitance and voltage
- Consider a CMOS NOT gate
- 0-1 transition at output draws energy from power source
  - ½ stored in capacitance
  - ½ dissipated as heat
- 1-0 transition doesn't draw energy from power source but ½ charge in capacitance dissipates as heat




$$E^{0-1}_{Dynamic} = \int_{v=0}^{v=V_{DD}} C\, v\, dv = \frac{1}{2} C\, {V_{DD}}^2 \quad \text{Joules}$$

$$E^{1-0}_{Dynamic} = \frac{1}{2} C\, {V_{DD}}^2 \quad \text{Joules}$$



# Circuit Power Consumption

- Power is energy consumed over time
  - 1 Watts = Amounts of Joules consumed in one second
- During each clock period some signals make 1-0 and some 0-1 transitions
- Energy consumed for all 0-1 transitions during one clock period:

$$E_{Dynamic} = \frac{1}{2} C_{Total}\, {V_{DD}}^2 \text{ Joules}$$

- Power consumed during one clock period

$$P_{Dynamic} = \frac{E_{Dynamic}}{\tau} = \frac{1}{2} C_{Total}\, {V_{DD}}^2 f \quad \text{watts}$$

- Signals not changing during clock period consume only static power

$$P_{Static} = V_{DD}\, I_{DD}$$

$I_{DD}$, DC or leakage current



# Some Ways to Reduce Dynamic Power Consumption

- Reduce total capacitance, $C_{Total}$
- Reduce supply voltage, $V_{DD}$
- Reduce clock frequency, $f$.
- Reduce glitches

$$P_{Dynamic} = \frac{1}{2} C_{Total}\, {V_{DD}}^2 f$$



# Power vs. Energy

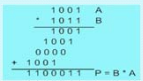
- Energy is independent of clock frequency
  - Increasing clock frequency increases power, not energy consumption
- Consider two CPUs A and B
  - Suppose A consumes 20% more power but executes a program quicker; requires %40 less time than B
- Which CPU is more energy efficient?

**CPU A uses 72% of energy used by CPU B**



# Unsigned Sequential Multiplier

- Hardware:
- Three registers A, B, and P
  - A and B are n-bits, P is n+1 bits
- A Mod n+1 counter (CNTR)
- An n-bit adder
- Basic algorithm:
  1. Initialization
     - Load values and clear register P and CNTR
  2. P ← P + A if next multiplier bit (b0) is 1
     - Skipping over 0 multiplier bits
  3. Right shift P //B, increment CNTR
  4. If not done go to 2
  5. Else, final product is in P//B



# Multiplier Algorithm

A ← A_value; B ← B_value; P ← 0; CNTR ← 0;
Do
    if(B[0] == 1)
        P ← P + A;
    {P, B} ← {P, B} >> 1;
    CNTR ← CNTR + 1;
While CNTR < n



# Design Entry Options
## (Data Path + FSD-based Controller)

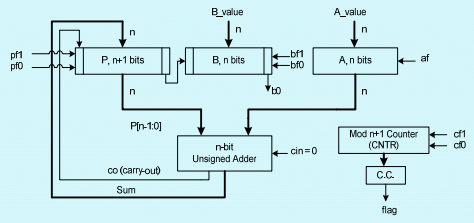
Option I: All structural
- Design all modules at gate level
- Recommended for schematic-only design tools

Option II: Hybrid
- Behavioral models for modules
- Combine modules to create structural model

Option III: All behavioral
- Behavioral model of NSG
- Behavioral model of OG
  - Describes data path
  - OG not combinational circuit in this case
- Behavioral model of FFs



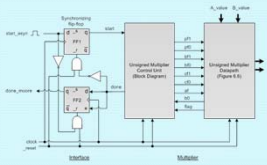
# Multiplier Data Path (Structural)






# Multiplier Controller (FSD)

- Assumes external input signal *start_asyn*
  - Uses one synchronizing FF (Ch5)
- Assumes external Moore output *done_moore*
  - indicates product is ready when asserted
- Draw controller FSD
  - E.g., Mealy





# Multiplier Model (All Behavioral)

- One module with five "always" blocks:
  1. Model of synchronization FF
  2. Model of *done*-Moore FF
  3. Combination circuit model of controller NSG
  4. Model of OG as data path (sequential circuit)
  5. Model of controller FFs



# Synchronizing FF

- Resets when *_reset* or *done* asserted
- Otherwise, loads asynchronous input *start_asyn*.

```
always@(posedge clock or negedge _reset or posedge done)
begin
        if(_reset == 0 || done == 1)
                start <= 1'b0;
        else
                start <= start_asyn;
end
```

# Converting done-Mealy to done-Moore

- Asynchronously de-asserts *done*-Moore when *_reset* or *start_asyn* asserted
- Synchronously asserts *done*-Moore when *done*-Mealy asserted

```
always@(posedge clock or negedge _reset or posedge start_asyn)
begin
    if(_reset == 0  || start_asyn == 1)
                done_moore = 1'b0;
    else
                done_moore <= done;  //done = 1 kept for one clock cycle
end
endmodule
```



# OG defines data path

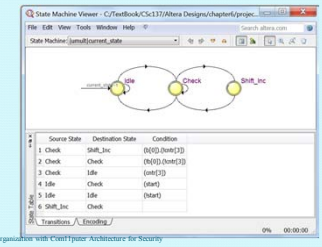
```
module umult(
    ....

assign result = {p[7:0], b};
//--------------- The states ----------------------------
------

parameter       Start =             2'b00,
                Check =             2'b01,
                Shift_Inc =         2'b10;

//----------------------- Output Generator (OG)
always@(posedge clock or _reset)
begin
    if(!_reset)
    begin

                p <= 0;
                cntr <= 0;
    end
    else
```

```
case(current_state)
    Start: if(start == 1)
                begin // initialize
                        a <= a_value;
                        b <= b_value;
                        p <= 0;
                        cntr <= 0;
            end
    Check:  begin
                if (cntr < 8)
                    if(b[0] == 1)
                        p <= p[7:0] + a;
                else
                begin
                    {p,  b}  <=  {p,  b}  >>  1;
                    cntr <= cntr + 1;
                end
            end
    Shift_Inc:  begin
                {p, b} <= {p, b} >> 1;
                cntr <= cntr + 1;
                end
    endcase
end
```



# FSD Verification

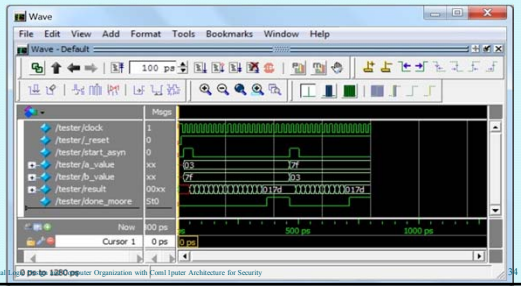
- Was the FSD description correct?
  - Tools can generates FSD from the HDL description for verification purposes





# Simulation Results

- 0x03 * 0x7f = 0x017d, longer multiplication time
- 0x7f * 0x03 = 0x017d, shorter multiplication time
- Optionally, can implement an initial switching circuit to switch A with B if A < B.






# 2's Complement Multiplier

- Multiplier B and multiplicand A are 2's complement numbers resulting in 2's complement product
- Basic data path operation is P ← P + A or P ← P – A
- Examines B bits two bits at a time overlapping
- Hardware:
  - Similar to unsigned multiplier except for adder/subtractor
- Algorithm:

A ← A_value[n-1]//A_value; B ← B_value//0; P ← 0; CNTR ← 0;
**Do**
  if(B[1] ⊕ B[0] == 1)  //overlapping two bits
    P ← P ± A;  //where m = B[1]
  {P, B} ← {P, B} >> 1;
  CNTR ← CNTR + 1;
**While CNTR < n**

- E.g., if B_value = $(1111)_{2s}$ = -1, then P = A * -1 = -A for arbitrary A_value

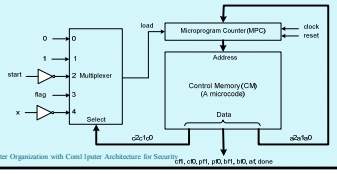


# 2's Complement Multiplier Microprogram Controller

- Multiplier microprogram

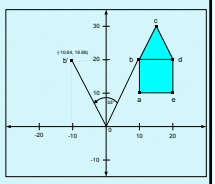
| | |
|---|---|
| 0 | If *start* == 0 go to 0; |
| 1 | P ← 0, A ← *A_value*, B ← *B_value*, CNTR ← 0; |
| 2 | If CNTR == n go to 6; |
| 3 | If B[1] ⊕ *B*[0]== 0 go to 5 |
| 4 | P ← P ± A; |
| 5 | {P, B} >>> 1, CNTR ← CNTR + 1, go to 2; |
| 6 | *done* = 1, go to 0; |

- Controller



# 2D Virtual Object Rotation

- Rotating object by β degrees requires rotating each vector by β degrees
  - $X' = \cos\beta * X - \sin\beta * Y$
  - $Y' = \sin\beta * X + \cos\beta * Y$
- Implemention requires cosine and sin functions
  - More hardware
- CORDIC algorithm simpler
  - Can use integer arithmetic
    - Addition, subtraction, and left shift for multiplication
  - Rotation done in steps
    - Factor out cos $\beta$
      - $X' = \cos\beta * (X - \tan\beta * Y)$
      - $Y' = \cos\beta * (\tan\beta * X + Y)$
    - E.g., 55° rotation is done in 4 stps:
      - 45°, 7°, 2°, 1°
    - Note, $\tan 45° = 1$, $\tan 7° = 1/8$, $\tan 2° = 1/32$, $\tan 1° = 1/64$
  - Done in steps results in a gain
    - E.g., for 55°, gain is $\frac{1}{\cos 47° * \cos 7° * \cos 2° * \cos 1°}$
    - Rotated object looks bigger





# CORDIC Rotation Algorithm

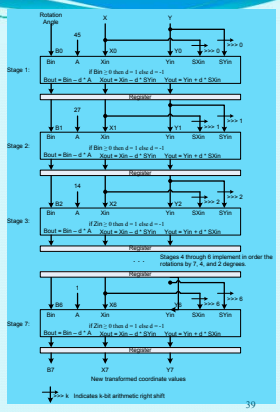
- Fix number of steps for $|\beta| \le 90°$
  - Rotation can be done in 7 steps
    - E.g., for 55°, steps are 45°, 27°, -14°, -7°, 4°, 2°, 1°
  - tan $\beta * X$ and tan $\beta * Y$ are computed by shifting X and Y by 0 bits in step 1, 2 bits in step 2, …, and 6 bits in step 7
- An initial rotation of 90° or 180° if $|\beta| > 90°$
- Replace $\beta$ with $\beta$ Mod 360° if $|\beta| > 360°$,
- Algorithm for $|\beta| \le 90°$:

```
If (β >= 0)
        d = 1;
else
        d = -1;
for (i = 0; i < 7, i++)
        x = x – d * (y >>> i);      //arithmetic right shift (integer division)
        y = d * (x >>> i) + y
        β = β – d * tan^-1 2^-i     //tan^-1 2^-i read from table or
                                    //hard coded as 45, 27, 14, 7, 4, 2, 1
endfor
```



# Pipelined CORDIC Rotation

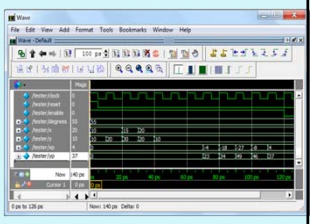
- Loop unrolling: 7 stages for 7 iterations of the for-loop
- Hard coded rotation angles
- Wired shifts implement integer division
- Simpler controller
  - All stages perform the same computations but operate on different data






# Simulation Result

- Illustrates pipeline vector transformation
- At 1 GHz clock, one billion peak transformations per second
- Typically million vectors per virtual object





# Rotated Virtual Object

- Shows a gain of 1.653 = 1/0.6048
  - CPU must multiply each new coordinate point by 0.6048 to get the actual size

    $0.6048 = \cos 47° * \cos 27° * \cos 14° * \cos 7° * \cos 2° * \cos 1°$
- New coordinate points include small error due to integer division used in the CORDIC algorithm