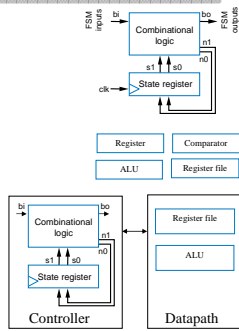


Lecture 5 Register-Transfer Level (RTL) Design

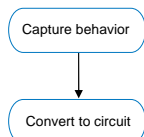
Introduction

- **Controllers**
 - Control input/output: single bit (or just a few) representing event or state
 - Finite-state machine describes behavior; implemented as state register and combinational logic
- **Datapath components**
 - Data input/output: Multiple bits collectively representing single entity
 - Datapath components included registers, adders, ALU, comparators, register files, etc.
- **Custom processors**
 - Controller and datapath components working together to implement an algorithm



RTL Design: Capture Behavior, Convert to Circuit

- **Combinational Logic Design**
 - First step: Capture behavior (using equation or truth table)
 - Remaining steps: Convert to circuit
- **Sequential Logic Design**
 - First step: Capture behavior (using FSM)
 - Remaining steps: Convert to circuit
- **RTL Design (the method for creating custom processors)**
 - First step: Capture behavior (using high-level state machine, to be introduced)
 - Remaining steps: Convert to circuit



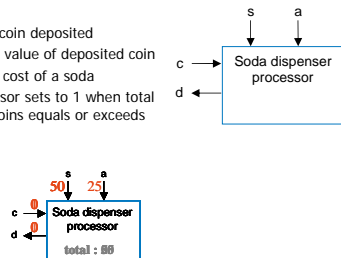
RTL Design Method

Step	Description
Step 1: Capture the high-level FSM	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 bit inputs and outputs
Step 2: Create a datapath	Create a datapath to carry out the data operations on the high-level state machine
Step 3: Connect the datapath to the controller	Connect the datapath to the controller block. Connect external Boolean inputs and output to the controller block
Step 4: Derive the controller's FSM	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

RTL Design Method

Soda Dispenser Example

- Soda dispenser
 - c : bit input, 1 when coin deposited
 - a : 8-bit input having value of deposited coin
 - s : 8-bit input having cost of a soda
 - d : bit output, processor sets to 1 when total value of deposited coins equals or exceeds cost of a soda

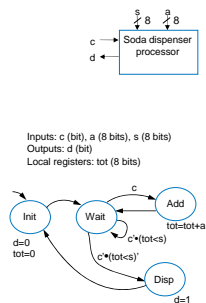


How can we precisely describe this processor's behavior?

RTL Design Method

Soda Dispenser - Step 1: Capture the high-level FSM

- Step 1: Describe behavior using high-level FSM
 - Start with inputs/output of system
 - Declare local register tot
 - **Init state**
 - Don't dispense soda ($d=0$), clear running total ($tot=0$)
 - **Wait state**
 - Wait for coin, if see coin go to Add state
 - **Add state**
 - Update total value: $tot = tot + a$
 - Remember, a is present coin's value
 - Go back to Wait state
 - **In Wait state**
 - If $tot \leq s$, Wait
 - If $tot > s$, go to Disp(ense) state
 - **Disp state**
 - Set $d=1$ (dispense soda)
 - Return to Init state



Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

- 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, displays computed distance

Digital Design Copyright © 2006 Frank Vahid
ECE 474a/575a Susan Lysdecky
13 of 91

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)

S_0 ?

$L = 0$ (turn laser off)
 $D = 0$ (set distance = 0)

- Step 1: Create high-level state machine
- Begin by declaring inputs and outputs
- Create initial state, name it S_0
 - Initialize laser to off ($L=0$)
 - Initialize displayed distance to 0 ($D=0$)

Digital Design Copyright © 2006 Frank Vahid
ECE 474a/575a Susan Lysdecky
14 of 91

Laser-Based Distance Measurer

Step 1 : Capture a high-level state machine

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)

S_0 S_1 S_2

$L = 0$
 $D = 0$

B' (button not pressed)
 B (button pressed)

- Add another state, call S_1 , that waits for a button press
 - B' – stay in S_1 , keep waiting
 - B – go to a new state S_2

Q: What should S_2 do?
A: Turn on the laser

Digital Design Copyright © 2006 Frank Vahid
ECE 474a/575a Susan Lysdecky
15 of 91

Laser-Based Distance Measurer

Step 2: Create a Datapath

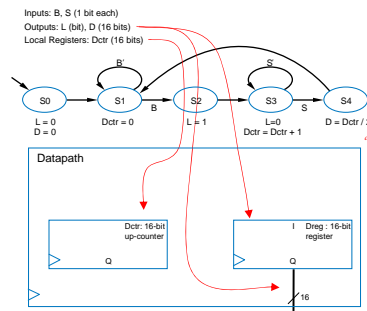
- Datapath must
 - Implement data storage
 - Implement data computations
- Look at high-level state machine, do three substeps
 - a) Make data inputs/outputs be datapath inputs/outputs
 - b) Instantiate declared registers into the datapath (also instantiate a register for each data output)
 - c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Instantiate: to introduce a new component into a design.

Laser-Based Distance Measurer

Step 2: Create a Datapath

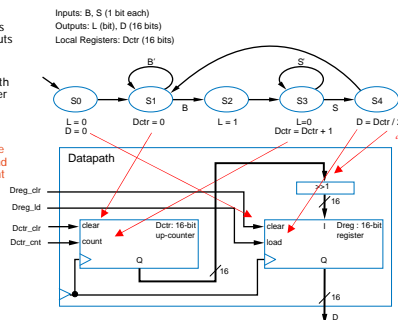
- a) **Make data inputs/outputs be datapath inputs/outputs**
- b) **Instantiate declared registers into the datapath (also instantiate a register for each data output)**
- c) **Examine every state and transition, and instantiate datapath components and connections to implement any data computations**



Laser-Based Distance Measurer

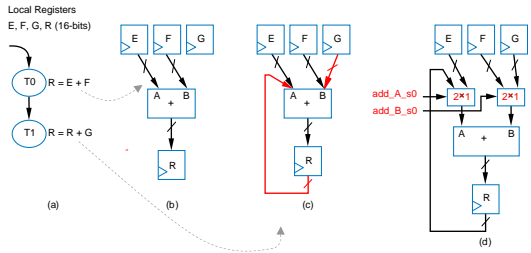
Step 2: Create a Datapath

- a) **Make data inputs/outputs be datapath inputs/outputs**
- b) **Instantiate declared registers into the datapath (also instantiate a register for each data output)**
- c) **Examine every state and transition, and instantiate datapath components and connections to implement any data computations**



Laser-Based Distance Measurer

Showing MUX Use

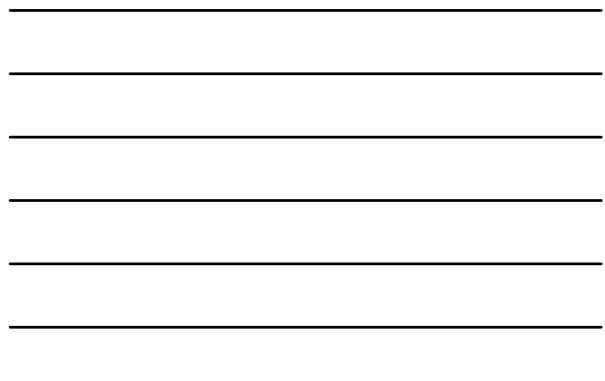
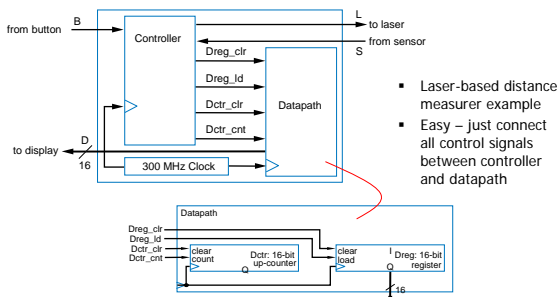


- Introduce mux when one component input can come from more than one source



Laser-Based Distance Measurer

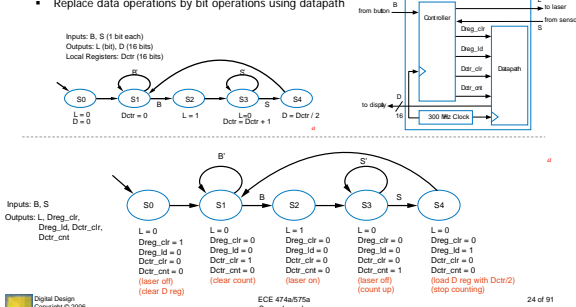
Step 3: Connecting the Datapath to a Controller



Laser-Based Distance Measurer

Step 4: Deriving the Controller's FSM

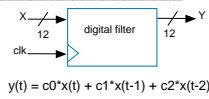
- FSM has same structure as high-level state machine
 - Inputs/outputs all bits now
 - Replace data operations by bit operations using datapath



Data Dominated RTL Design Example

FIR Filter

- FIR filter
 - "Finite Impulse Response"
 - Simply 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 specific filter



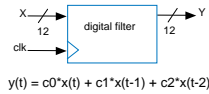
$$y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-1) + c_2 \cdot x(t-2)$$

- Step 1: Create high-level state machine
 - But there really is none! Data dominated indeed.
- Go straight to step 2

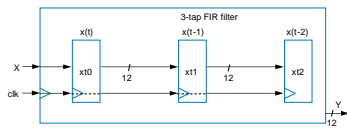
Data Dominated RTL Design Example

FIR Filter

- Step 2: Create datapath
 - Begin by creating chain of $x(t)$ registers to hold past values of X



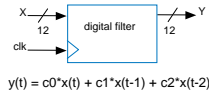
$$y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-1) + c_2 \cdot x(t-2)$$



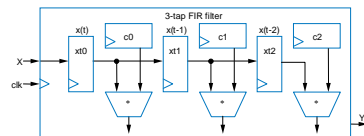
Data Dominated RTL Design Example

FIR Filter

- Step 2: Create datapath (cont.)
 - Instantiate registers for c_0 , c_1 , c_2
 - Instantiate multipliers to compute $c \cdot x$ values

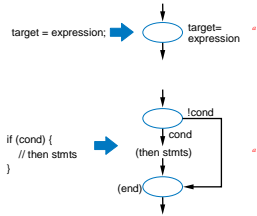


$$y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-1) + c_2 \cdot x(t-2)$$



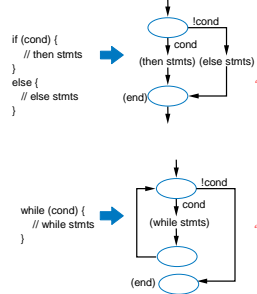
Converting from C to High-Level State Machine

- 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



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

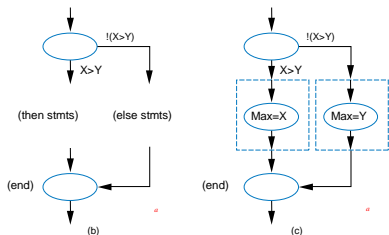


Simple Example of Converting from C to High-Level State Machine

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

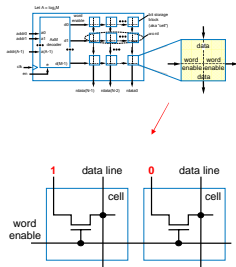
Inputs: uint X, Y
Outputs: uint Max

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



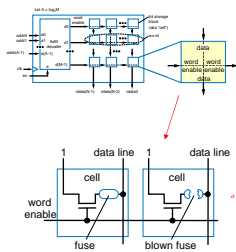
ROM Types

- If a ROM can only be read, how are the stored bits stored in the first place?
 - Storing bits in a ROM known as *programming*
 - Several methods
- **Mask-programmed ROM**
 - Bits are hardwired as 0s or 1s during chip manufacturing
 - 2-bit word on right stores "10"
 - word enable (from decoder) simply passes the hardwired value through transistor
 - Notice how compact, and fast, this memory would be



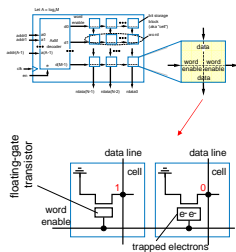
ROM Types

- **Fuse-Based Programmable ROM**
 - Each cell has a fuse
 - A special device, known as a programmer, blows certain fuses (using higher-than-normal voltage)
 - Those cells will be read as 0s (involving some special electronics)
 - Cells with unblown fuses will be read as 1s
 - 2-bit word on right stores "10"
- Also known as **One-Time Programmable (OTP) ROM**



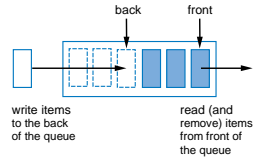
ROM Types

- **Erasable Programmable ROM (EPROM)**
 - Uses "floating-gate transistor" in each cell
 - Special programmer device uses higher-than-normal voltage to cause electrons to tunnel into the gate
 - Electrons become trapped in the gate
 - Only done for cells that should store 0
 - Other cells (without electrons trapped in gate) will be 1
 - 2-bit word on right stores "10"
 - Details beyond our scope – just general idea is necessary here
 - To erase, shine ultraviolet light onto chip
 - Gives trapped electrons energy to escape
 - Requires chip package to have window



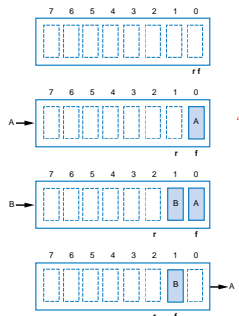
Queues

- A queue is another component sometimes used during RTL design
- Queue:** A list written to at the back, from read from the front
 - Like a list of waiting restaurant customers
- Writing called a **push**, reading called a **pop**
- Because first item written into a queue will be the first item read out, also called a **FIFO** (first-in-first-out)



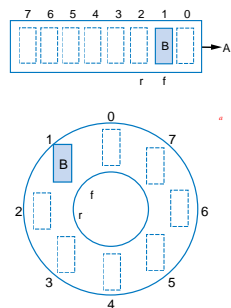
Queues

- Queue has addresses, and two pointers: **rear** and **front**
 - Initially both point to 0
- Push (write)
 - Item written to address pointed to by **rear**
 - rear** incremented
- Pop (read)
 - Item read from address pointed to by **front**
 - front** incremented
- If front or rear reaches 7, next (incremented) value should be 0 (for a queue with addresses 0 to 7)



Queues

- Treat memory as a circle
 - If front or rear reaches 7, next (incremented) value should be 0 rather than 8 (for a queue with addresses 0 to 7)
- Two conditions of interest
 - Full queue – no room for more items
 - In 8-entry queue, means 8 items present
 - No further pushes allowed until a pop occurs
 - Causes front=rear
 - Empty queue – no items
 - No pops allowed until a push occurs
 - Causes front=rear
- Both conditions have front=rear
 - To detect whether front=rear means full or empty, need state machine that detects if previous operation was push or pop, sets full or empty output signal (respectively)



Summary

- Modern digital design involves creating processor-level components
- Four-step RTL method can be used
 1. High-level state machine
 2. Create datapath
 3. Connect datapath to controller
 4. Derive controller FSM
- Several examples
 - Control dominated, data dominated, and mix
- Determining fastest clock frequency
 - By finding critical path
- Behavioral-level design – C to gates
 - By using method to convert C (subset) to high-level state machine
- Additional RTL components
 - Memory: RAM, ROM
 - Queues
