Components and Sequential Circuits

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

February 27, 2020
Overview

- Vending machine project
- Repeat combinational building blocks
- Power user II
- Components and top-level
- Sequential circuits
How is the lab work going so far?
Start to organize yourself in groups of 2–3
  You can ask for finding a group via slack
Register at Google spreadsheet
A Vending Machine from 1952

Source: Minnesota Historical Society, CC BY-SA 2.0
The Vending Machine

- Final project is a vending machine
- *Specification document is in DTU Inside (show it)*
- Inputs: coins, buy
- Display: price and current amount
- Output: release can or error
- Small challenge to multiplex the display
- State machine with data path is the *brain* of the VM
- Guided step by step over several weeks
Vending Machine Specification I

- Sell 1 item and not returning any money
- Set price with 5 switches (1–31 kr.)
- Display price on two 7-segment displays (hex.)
- Accept 2 and 5 kr. (two push buttons)
- Display sum on two 7-segment displays (hex.)
  - Amount entered so far
- Does not return money, left for the next purchase
Vending Machine Specification II

- Push button *Buy*
  - If not enough money, activate *alarm* as long as *buy* is pressed
  - If enough money, activate *release item* for as long as *buy* is pressed and reduce *sum* by the price of the item

- Optional extras (for a 12)
  - Display decimal numbers
  - Supplement alarm by some visuals (e.g., blinking display)
  - Count coins and display an alarm when compartment is full (> 20 coins)
  - Have some text scrolling on the display
  - ...
  - Your ideas :-(
Design and Implementation

- Implementation shall be a state machine plus datapath
- Design your datapath on a sheet of paper

Datapath
- Does add and subtract
- Contains a register to hold the sum
- Needs some multiplexer to operate

Display needs multiplexing
- Implemented with some counters and a multiplexer

Show each part of your design to a TA
- 7-segment decoder, 7-segment with a counter, display multiplexer, complete vending machine
Vending Machine Design and Implementation Steps

- We start next week
- $2 + 2 + 3 + 3 + 4 + 4 = 18$ supervised lab hours
- 1a. Hexadecimal to 7-segment decoder
- 1b. 7-segment display with a counter
- 2. Multiplexed Seven-Segment Display
- 3. Complete Vending Machine
- *Show your working design to a TA*
Final Report

- One report per group
- A single PDF
  - Your group number is part of the file name (e.g., group7.pdf)
  - Code as listing in an appendix (no .zip files)
  - Hand in in DTU Inside
- Content
  - Abstract
  - Preface (Who did what)
  - 1. Introduction and Problem Formulation
  - 2. Analysis and Design
  - 3. Implementation
  - 4. Testing
  - 5. Results
  - 6. Discussion
  - 7. Conclusion
  - List of References
  - Appendix: Chisel code
Questions on Final Project?
Combinational Circuit with Conditional Update

- Value first needs to be wrapped into a Wire
- Updates with the Chisel update operation :=
- With when we can express a conditional update
- The condition is an expression with a Boolean result
- The resulting circuit is a multiplexer
- The rule is that the last enabled assignment counts
  - Here the order of statements has a meaning

```scala
val enoughMoney = Wire(Bool())

enoughMoney := false.B
when (coinSum >= price) {
  enoughMoney := true.B
}
```
Comparison

- The usual operations (as in Java or C)
  - Unusual equal and unequal operator symbols
  - To keep the original Sala operators usable for references
- Operands are UInt and SInt
- Operands can be Bool for equal and unequal
- Result is Bool

>, >=, <, <=
===, /=
Boolean Logical Operations

- Operands and result are `Bool`
- Logical NOT, AND, and OR

```scala
val notX = !x
val bothTrue = a && b
val orVal = x || y
```
The “Else” Branch

- We can express a form of “else”
- Note the . in .otherwise

```scala
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .otherwise {
  w := 2.U
}
```
A Chain of Conditions

- To test for different conditions
- Select with a priority order
- The first expression that is true counts
- The hardware is a chain of multiplexers

```scala
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .elseif (cond2) {
  w := 2.U
} .otherwise {
  w := 3.U
}
```
Default Assignment

- Practical for complex expressions
- Forgetting to assign a value on all conditions
  - Would describe a latch
  - Runtime error in Chisel
- Assign a default value is good practise

```scala
val w = WireDefault(0.U)
when (cond) {
  w := 3.U
}
// ... and some more complex conditional assignments
```
Logic Can Be Expressed as a Table

- Sometimes more convenient
- Still combinational logic (gates)
- Is converted to Boolean expressions
- Let the synthesize tool do the conversion!
- We use the switch statement

```c
switch (sel) {
    is ("b00".U) { result := "b0001".U}
    is ("b01".U) { result := "b0010".U}
    is ("b10".U) { result := "b0100".U}
    is ("b11".U) { result := "b1000".U}
}
```
A Decoder

- Converts a binary number of \( n \) bits to an \( m \)-bit signal, where \( m \leq 2^n \)
- The output is one-hot encoded (exactly one bit is one)
- Building block for a \( m \)-way Mux
- Used for address decoding in a computer system
- Maybe of use for the display multiplexer
## Truth Table of a Decoder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>01</td>
<td>0010</td>
</tr>
<tr>
<td>10</td>
<td>0100</td>
</tr>
<tr>
<td>11</td>
<td>1000</td>
</tr>
</tbody>
</table>
An Encoder

▶ Converts one-hot encoded signal
▶ To binary representation
### Truth Table of an Encoder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>00</td>
</tr>
<tr>
<td>0010</td>
<td>01</td>
</tr>
<tr>
<td>0100</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
</tr>
<tr>
<td>?? ??</td>
<td>??</td>
</tr>
</tbody>
</table>

- Only defined for one-hot input
We cannot describe a function with undefined outputs

We use a default assignment of "b00"

```java
b := "b00".U
switch (a) {
    is ("b0001".U) { b := "b00".U}
    is ("b0010".U) { b := "b01".U}
    is ("b0100".U) { b := "b10".U}
    is ("b1000".U) { b := "b11".U}
}
```
Every craftsmen starts with good-quality tools
“Tools amplify your talent”¹
  ▶ The better your tools, the more productive you are
  ▶ The better you know them, the more productive you are
IDEs (Eclipse, InelliJ) are nice, I love them too
But we shall go beyond it
Use tools (and write your own)
Help with: google, man pages, or even plain –help (or -h)
https://www.oreilly.com/learning/ten-steps-to-linux-survival
  ▶ This is about command line tools, not just Linux

¹The Pragmatic Programmer: From Journeyman to Master, by Andrew Hunt and David Thomas
Use the command line, shell, terminal

In Windows: PowerShell
  - You may want to install the Linux subsystem

Universal Unix commands (Windows, Mac, Linux)

Navigating the file system:
  - Change directory: `cd`
  - Print working directory: `pwd`
  - Make a directory: `mkdir abc`
  - Create a file: `echo test > abc.txt`
  - Show file content: `cat abc.txt`
  - Remove a file: `rm abc.txt`

Run your Chisel code with `sbt run`

You used the terminal already from within IntelliJ ;-)

We talked about git last week
To version your source
Maybe hosting on GitHub
Most teaching material is on GitHub
Use `git pull` to update the lab material
Show how to use it, now!
- Clone a repo: `git clone path`
- Get the newest version: `git pull`
- Further commands: `git commit`, `push`, `log`, `status`
- Overview of changes: `gitk`
Structure With Bundles

- A Bundle to group signals
- Can be different types
- Defined by a class that extends Bundle
- Named fields as vals within the block
- Like a C struct or VHDL record

```scala
class Channel() extends Bundle {
  val data = UInt(32.W)
  val valid = Bool()
}
```
Using a Bundle

- Create it with `new`
- Wrap it into a `Wire`
- Field access with `dot` notation

```scala
val ch = Wire(new Channel())
ch.data := 123.U
ch.valid := true.B

val b = ch.valid
```
Components

- Components are building blocks
- Components have input and output ports (= pins)
  - Organized as a Bundle
  - assigned to field io
- We build circuits as a hierarchy of components
- In Chisel a component is called Module
- Components/Modules are used to organize the circuit
  - Similar as using methods in Java
Hierarchy of Components Example

- CompA
- CompB
- CompC
- CompD
Input/Output Ports

- Ports are bundles with directions
- Ports used to connect modules

class AluIO extends Bundle {
    val function = Input(UInt(2.W))
    val inputA = Input(UInt(4.W))
    val inputB = Input(UInt(4.W))
    val result = Output(UInt(4.W))
}
An Adder Module

- A class that extends Module
- Interface (port) is a Bundle, wrapped into an IO(), and stored in the field io
- Circuit description in the constructor

```scala
class Adder extends Module {
  val io = IO{
    new Bundle {
      val a = Input(UInt(4.W))
      val b = Input(UInt(4.W))
      val result = Output(UInt(4.W))
    }
  }

  val addVal = io.a + io.b
  io.result := addVal
}
```
Connections

- Simple connections just with assignments, e.g.,

  \[
  \text{adder.io.a := ina} \\
  \text{adder.io.b := inb}
  \]

- Note the dot access to the field \text{io} and then the IO field
Module Usage

- Create with `new` and wrap into a `Module()`
- Interface port via the `io` field
- Note the assignment operator `:=` on `io` fields

```scala
val adder = Module(new Adder())
adder.io.a := ina
adder.io.b := inb
val result = adder.io.result
```
Example: Arithmetic Logic Unit

- Also called ALU
- A central component of a microprocessor
- Two inputs, one function select, and an output
- Part of the *datapath*
Example: Arithmetic Logic Unit

class Alu extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(16.W))
        val b = Input(UInt(16.W))
        val fn = Input(UInt(2.W))
        val y = Output(UInt(16.W))
    })

    // some default value is needed
    io.y := 0.U

    // The ALU selection
    switch(io.fn) {
        is(0.U) { io.y := io.a + io.b }
        is(1.U) { io.y := io.a - io.b }
        is(2.U) { io.y := io.a | io.b }
        is(3.U) { io.y := io.a & io.b }
    }
}
Hierarchy of Components Example

CompA

CompB

CompC

CompD
Components CompA and CompB

class CompA extends Module {
    val io = IO(new Bundle {
        val a = Input(UInt(8.W))
        val b = Input(UInt(8.W))
        val x = Output(UInt(8.W))
        val y = Output(UInt(8.W))
    })

    // function of A
}

class CompB extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })

    // function of B
}
Component CompC

class CompC extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_x = Output(UInt(8.W))
    val out_y = Output(UInt(8.W))
  })

  // create components A and B
  val compA = Module(new CompA())
  val compB = Module(new CompB())

  // connect A
  compA.io.a := io.in_a
  compA.io.b := io.in_b
  io.out_x := compA.io.x

  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.in_c
Create one top-level Module

Invoke the Chisel driver from the App

Pass the top module (e.g., `new Hello()`)

Optional: pass some parameters (in the Array)

Following code generates Verilog code for *Hello World*

```java
object Hello extends App {
    chisel3.Driver.execute(Array[String](), () =>
        new Hello())
}
```
class Hello extends Module {
  val io = IO(new Bundle {
    val led = Output(UInt(1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U;

  val cntReg = RegInit(0.U(32.W))
  val blkReg = RegInit(0.U(1.W))

  cntReg := cntReg + 1.U
  when(cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}

Hello is the top-level of our blinking LED
No real need to read this code
But pin assignment for the synthesis
Additional pins: clock and reset
User pin names with a leading io:

module Hello(
    input   clock,
    input   reset,
    output  io_led
);

We can find our two register definitions

@... gives Chisel source and line number (e.g., 17)

```verilog
reg [31:0] cntReg; // @[Hello.scala 17:23]
reg [31:0] _RAND_0;
reg blkReg; // @[Hello.scala 18:23]
```
The increment and comparison against maximum value

```verilog
assign _T_1 = cntReg + 32'h1; // @Hello.scala 20:20
assign _T_2 = cntReg == 32'h2faf07f; // @Hello.scala 21:15
assign _T_3 = ~ blkReg; // @Hello.scala 23:15
assign io_led = blkReg; // @Hello.scala 25:10
```
Verilog register code

always @(posedge clock) begin
    if (reset) begin
        cntReg <= 32'h0;
    end else if (_T_2) begin
        cntReg <= 32'h0;
    end else begin
        cntReg <= _T_1;
    end
end
Verilog Generation Summary

- Verilog is generated for synthesis
- We do not need to read it
- Just pins are interesting
- Additional clock and reset
- Pin names with additional io
A Scala file can contain several classes (and objects)
For large classes use one file per class with the class name
Scala has packages, like Java
Use folders with the package names for file organization
sbt looks into current folder and src/main/scala/
Tests shall be in src/test/scala/
File Organization in Scala/Chisel

```
project
  src
    main
      scala
        package
          sub-package
    test
      scala
        package
  target
    generated
```
What is a Minimal Chisel Project?

- Scala class (e.g., Hello.scala)
- Build info in build.sbt for sbt:

```scala
scalaVersion := "2.11.12"

libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.2.2"
```
Sequential Building Blocks

- Contain a register
- Plus combinational circuits

\[
\text{val } q = \text{RegNext}(d)
\]
Register With Reset

```scala
val valReg = RegInit(0.U(4.W))
valReg := inVal
```
Timing Diagram of the Register with Reset

- Also called waveform diagram
- Logic function over time
- Can be used to describe a circuit function
- Useful for debugging
Register with Enable

▶ Only when enable true is a value is stored

```scala
val enableReg = Reg(UInt(4.W))

when (enable) {
    enableReg := inVal
}
```
A Register with Reset and Enable

- We can combine initialization and enable

```scala
val resetEnableReg = RegInit(0.U(4.W))
when (enable) {
  resetEnableReg := inVal
}
```

- A register can also be part of an expression
- What does the following circuit do?

```scala
val risingEdge = din & !RegNext(din)
```
A Register with an Adder is a Counter

- Is a free running counter
- 0, 1, ... 14, 15, 0, 1, ...

```scala
def cntReg = RegInit(0.U(4.W))
cntReg := cntReg + 1.U
```
A Counter with a Mux

```scala
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```

- This counter counts from 0 to 9
- And starts from 0 again after reaching 9
  - Starting from 0 is common in computer engineering
- A counter is the hardware version of a `for loop`
- Often needed
val cntEventsReg = RegInit(0.U(4.W))
when(event) {
  cntEventsReg := cntEventsReg + 1.U
}

Counting Events
Counting Up and Down

▶ Up:

\[
\text{val cntReg} = \text{RegInit}(0.U(8.W))
\]

\[
\text{cntReg} := \text{cntReg} + 1.U
\]

\[
\text{when}(\text{cntReg} === N) \{
    \text{cntReg} := 0.U
\}
\]

▶ Down:

\[
\text{val cntReg} = \text{RegInit}(N)
\]

\[
\text{cntReg} := \text{cntReg} - 1.U
\]

\[
\text{when}(\text{cntReg} === 0.U) \{
    \text{cntReg} := N
\}\]
Testing with Chisel

- Tester extends class PeekPokeTester
- Has the device under test (DUT) as parameter
- Testing code can use all features of Scala

```scala
class CounterTester(dut: Counter) extends PeekPokeTester(dut) {

  // Here comes the Chisel/Scala code
  // for the testing
}
```
Testing

- Set input values with `poke`
- Advance the simulation with `step`
- Read the output values with `peek`
- Compare the values with `expect`
Testing Example

// Set input values
poke(dut.io.a, 3)
poke(dut.io.b, 4)
// Execute one iteration
step(1)
// Print the result
val res = peek(dut.io.result)
println(res)

// Or compare against expected value
expect(dut.io.result, 7)
Chisel Main for Testing

- Tests can be written in Scala/Chisel
- Invoke `execute` with some parameters, the DUT, and a tester

```scala
object CounterTester extends App {

    iotesters.Driver.execute(Array[String](), () =>
        new Counter(2)) {
            c => new CounterTester(c)
        }
}
```

- More on testing and waveform generation next week
Common Acronyms

ADC  analog-to-digital converter
ALU  arithmetic and logic unit
ASIC application-specific integrated circuit
Chisel constructing hardware in a Scala embedded language
CISC complex instruction set computer
CRC cyclic redundancy check
DAC digital-to-analog converter
DFF D flip-flop, data flip-flop
DMA direct memory access
DRAM dynamic random access memory
FF  flip-flop
Common Acronyms II

FIFO  first-in, first-out
FPGA  field-programmable gate array
HDL   hardware description language
HLS   high-level synthesis
IC    instruction count
IDE   integrated development environment
IO    input/output
ISA   instruction set architecture
JDK   Java development kit
JIT   just-in-time
JVM   Java virtual machine
LC    logic cell
Common Acronyms III

- **LRU**  least-recently used
- **MMIO**  memory-mapped IO
- **MUX**  multiplexer
- **OO**  object oriented
- **RISC**  reduced instruction set computer
- **SDRAM**  synchronous DRAM
- **SRAM**  static random access memory
- **TOS**  top-of stack
- **UART**  universal asynchronous receiver/transmitter
- **VHDL**  VHSIC hardware description language
- **VHSIC**  very high speed integrated circuit
Lab Today

- Components and Small Sequential Circuits
- Lab 4 Page
- You need to download again, as I have updated the lab
  - Or learn to use git and do a `git pull` ;-)
- Each exercise contains a test, which initially fails
- `sbt test` runs them all
  - To just run a single test, run e.g.,
    - `sbt "testOnly SingleTest"
- When all test succeed your are done ;-)
- Except: additional some drawing exercise
Summary

- Vending machine is your final project
- The vending machine and the report are part of your grade
- A digital circuit is organized in components
- Components have ports with directions
- Sequential circuits are combinations of registers with combinational circuits