Components and Sequential Circuits

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

February 15, 2024

No Lecture in Week 5

- In two weeks (29/2)
- No lecture, but self study of timing material
- Paper and pencil exercises in the lab
- I am in a teaching workshop from our section (ESE)

Overview

- Vending machine project
- Repeat combinational building blocks
- Power user II
- Components and top-level
- Sequential circuits

Admin

- How is the lab work going so far? Too easy?
- Continue to organize yourself in groups of 2–3
  - 1 is also OK
  - You can ask for finding a group via slack (in channel general)
- There is a group defined in Learn to register
  - You have to show parts of the Vending Machine to a TA
  - In the week that follows the exercise
  - On time: full points, one week late: half the points

A Vending Machine from 1952

Source: Minnesota Historical Society, CC BY-SA 2.0

The Vending Machine

- Final project is a vending machine
- Detailed specification document will be given
  - Put into the public in chisel-lab
- 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
- Accept 2 and 5 kr. (two push buttons)
- Display sum on two 7-segment displays
  - Amount entered so far
- Does not return money, left for the next purchase
- Display decimal numbers

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)
  - 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
  - Supplement alarm with some audio
  - Talk to the user (via serial port)
  - ...
  - 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 in week 6
  - Hexadecimal to 7-segment decoder
  - 7-segment display with a counter
  - Multiplexed Seven-Segment Display
  - Testing the Vending Machine
  - Complete Vending Machine
- *Show steps and your final 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 Learn
- 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

```
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

```
val notX = !x
val bothTrue = a && b
val orVal = x || y
```

The “Else” Branch

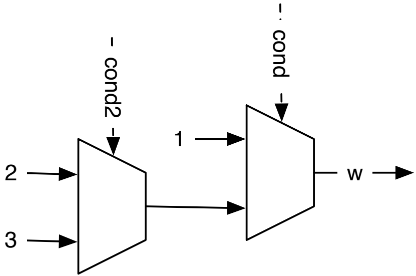
- We can express a form of “else”
- Note the `.` in `.otherwise`

```
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

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

when (cond) {
  w := 1.U
} .elsewhen (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

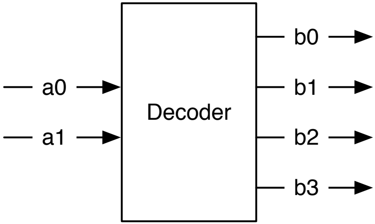
```
val w = WireDefault(0.U)

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

```
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

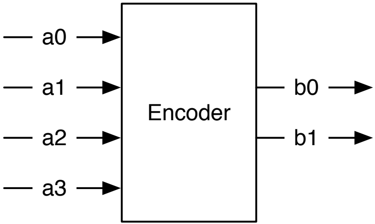


- Converts a binary number of n bits to an m-bit signal, where $m \leq 2^n$
- The output is one-hot encoded (exactly one bit is one)
- Building block for a m-way Mux
- Used for address decoding in a computer system
- Maybe of use for the display multiplexer

Truth Table of a Decoder

a	b
00	0001
01	0010
10	0100
11	1000

An Encoder



- Converts one-hot encoded signal
- To binary representation

Truth Table of an Encoder

a	b
0001	00
0010	01
0100	10
1000	11
????	??

- Only defined for one-hot input

Encoder in Chisel

- We cannot describe a function with undefined outputs
- We use a default assignment of `"b00"`

```
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}
}
```

Power User II

- 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

[1] The Pragmatic Programmer: From Journeyman to Master, by Andrew Hunt and David Thomas

Power User II

- 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 ;-)

Power User II

- 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`
- There are also GUI tools available, IntelliJ includes git support

Structure With Bundles

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

```
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

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

val b = ch.valid
```

Wire, Reg, and IO

- `UInt`, `SInt`, and `Bits` are Chisel types, not hardware
- `Wire`, `Reg`, or `IO` generates hardware
  - A `Wire` is a combinational circuit
  - A `Reg` is a register
  - A `IO` is a connection (for a module)
- Can wrap any Chisel type, also `Bundle` or `Vec`
- Give it a name by assigning it to a `val`

```
val number = Wire(UInt())
val reg = Reg(SInt())
```

Using = or :=

- Later assign or reassign a value or expression with `:=`

```
number := 10.U
reg := value - 3.U
```

- Note the small difference between = and `:=`
  - May be confusing to start with
- Use = when *creating* a hardware object to give it a name
- Use `:=` when assigning or reassigning to an *existing* hardware object

Components/Modules

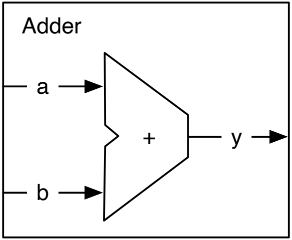
- Components/Modules are building blocks
  - Component and module are two names for the same thing
- Components have input and output ports (= pins)
  - Organized as a `Bundle`
  - Wrapped into an `IO()`
  - assigned to a field `io`
- We build circuits as a hierarchy of components
- In Chisel a component is called `Module`
- Components/Modules are used to organize the circuit
  - Similar to classes and methods in Java

Input/Output Ports

- Ports are the *interface* to a module
- Ports are bundles with directions
- Ports are 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



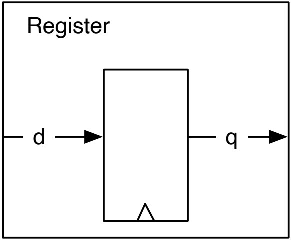
- Practically too simple, but for the slides

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

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

  io.y := io.a + io.b
}
```

An Register Module

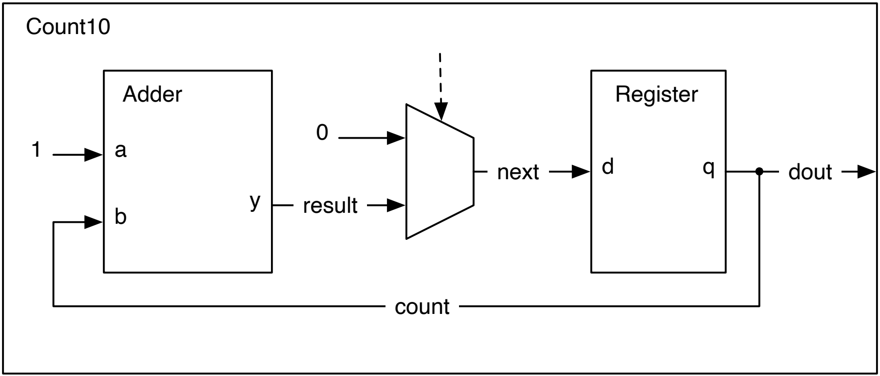


- Practically too simple, but for the slides

An Register Module

```
class Register extends Module {
  val io = IO(new Bundle {
    val d = Input(UInt(8.W))
    val q = Output(UInt(8.W))
  })

  val reg = RegInit(0.U)
  reg := io.d
  io.q := reg
}
```

An Counter out of Modules



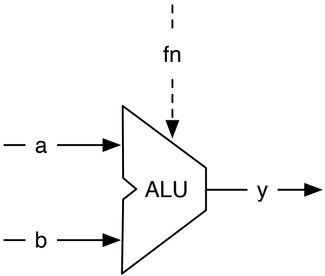
An Counter out of Modules

```
class Count10 extends Module {
  val io = IO(new Bundle {
    val dout = Output(UInt(8.W))
  })

  val add = Module(new Adder())
  val reg = Module(new Register())

  // the register output
  val count = reg.io.q
  // connect the adder
  add.io.a := 1.U
  add.io.b := count
  val result = add.io.y
  // connect the Mux and the register input
  val next = Mux(count === 9.U, 0.U, result)
  reg.io.d := next
  io.dout := count
}
```

Using Modules/Components

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

Example: Arithmetic Logic Unit

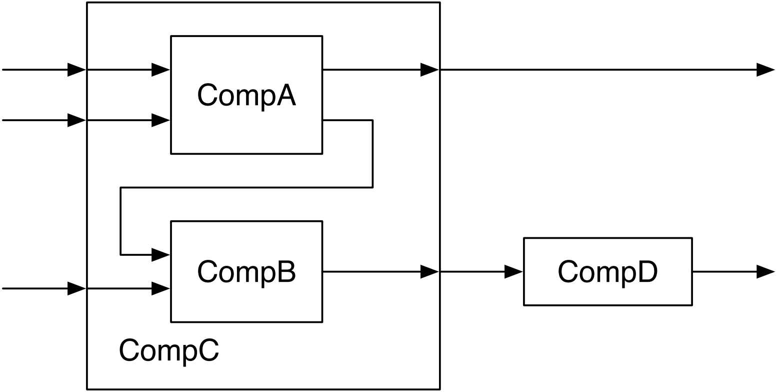


- Also called ALU
- A central component of a microprocessor
- Two inputs, one function select, and an output
- Part of the *datapath*

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 }
  }
}
```

Nested Components Example



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 inA = Input(UInt(8.W))
    val inB = Input(UInt(8.W))
    val inC = Input(UInt(8.W))
    val outX = Output(UInt(8.W))
    val outY = 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.inA
  compA.io.b := io.inB
  io.outX := compA.io.x
  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.inC
```

Chisel Main

- Create one top-level Module
- Invoke the Chisel code emitter from the App
- Pass the top module (e.g., `new Hello()`)
- Optional: pass some parameters (in an `Array`)
- Following code generates Verilog code for *Hello World*

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

Hello World in Chisel

```
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
}
```

Generated Verilog for Hello

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

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

Generated Verilog for Hello

- We can find our two register definitions
- `@...` gives Chisel source and line number (e.g., 17)

```
reg [31:0] cntReg; // @[Hello.scala 17:23]
reg  blkReg; // @[Hello.scala 18:23]
```

Generated Verilog for Hello

- The increment and IO connection

```
wire [31:0] _cntReg_T_1 = cntReg + 32'h1; // @[Hello.scal
assign io_led = blkReg; // @[Hello.scala 25:10]
```

Generated Verilog for Hello

- Verilog register code, including comparison against maximum value

```
always @(posedge clock) begin
  if (reset) begin // @[Hello.scala 17:23]
    cntReg <= 32'h0; // @[Hello.scala 17:23]
  end else if (cntReg == 32'h2faf07f) begin // @[Hello.sc
    cntReg <= 32'h0; // @[Hello.scala 22:12]
  end else begin
    cntReg <= _cntReg_T_1; // @[Hello.scala 20:10]
  end
  if (reset) begin // @[Hello.scala 18:23]
    blkReg <= 1'h0; // @[Hello.scala 18:23]
  end else if (cntReg == 32'h2faf07f) begin // @[Hello.sc
    blkReg <= ~blkReg; // @[Hello.scala 23:12]
  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_`

File Organization in Scala/Chisel

- 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`:

```
scalaVersion := "2.12.13"

scalacOptions ++= Seq(
  "-feature",
  "-language:reflectiveCalls",
)
```

Minimal Chisel Project Cont.

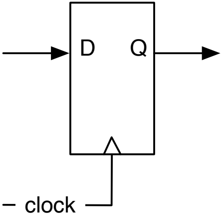
```
// Chisel 3.5
addCompilerPlugin("edu.berkeley.cs" %
   "chisel3-plugin" % "3.5.0" cross
   CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %%
   "chisel3" % "3.5.0"
libraryDependencies += "edu.berkeley.cs" %%
   "chiseltest" % "0.5.0"
```

Show It

- The absolute minimum is two files
  - `build.sbt`
  - A single `.scala` file

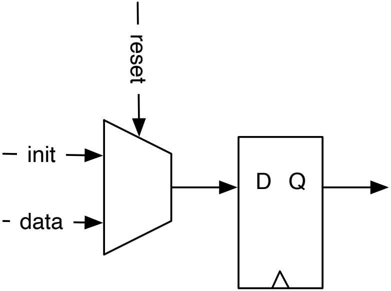
Sequential Building Blocks

- Contain a register
- Plus combinational circuits



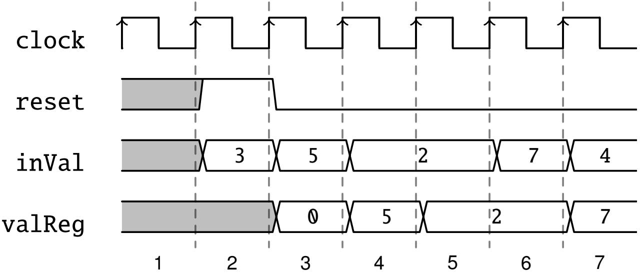

```
val q = RegNext(d)
```

Register With Reset



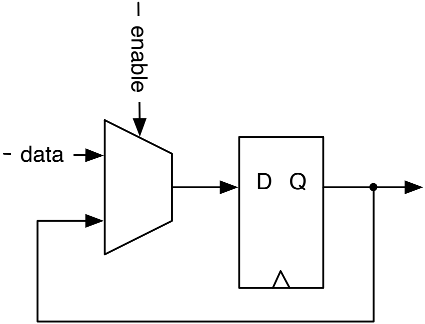

```
val valReg = RegInit(0.U(4.W))

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

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

when (enable) {
  enableReg := inVal
}
```

A Register with Reset and Enable

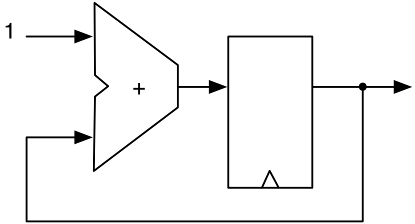
- We can combine initialization and enable

```
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?

```
val risingEdge = din & !RegNext(din)
```

A Register with an Adder is a Counter



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

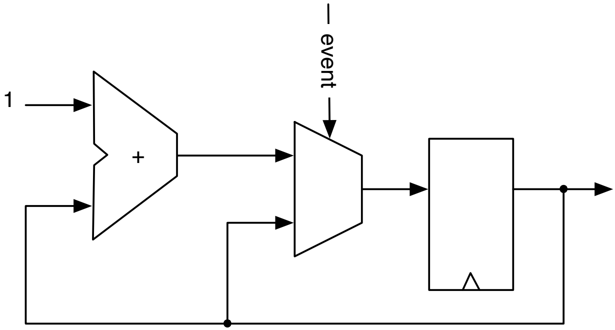
```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
```

A Counter with a Mux

```
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

Counting Events




```
val cntEventsReg = RegInit(0.U(4.W))
when(event) {
  cntEventsReg := cntEventsReg + 1.U
}
```

Counting Up and Down

- Up:

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

cntReg := cntReg + 1.U
when(cntReg === N) {
  cntReg := 0.U
}
```

- Down:

```
val cntReg = RegInit(N)

cntReg := cntReg - 1.U
when(cntReg === 0.U) {
  cntReg := N
}
```

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-Iin-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 3 Page
- 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 tests succeed your are (almost) done ;-)
- Additional some drawing exercise
- Do them, they will be part of the exam!

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