Introduction to Chisel

Xiangdong Wei

JI VE470

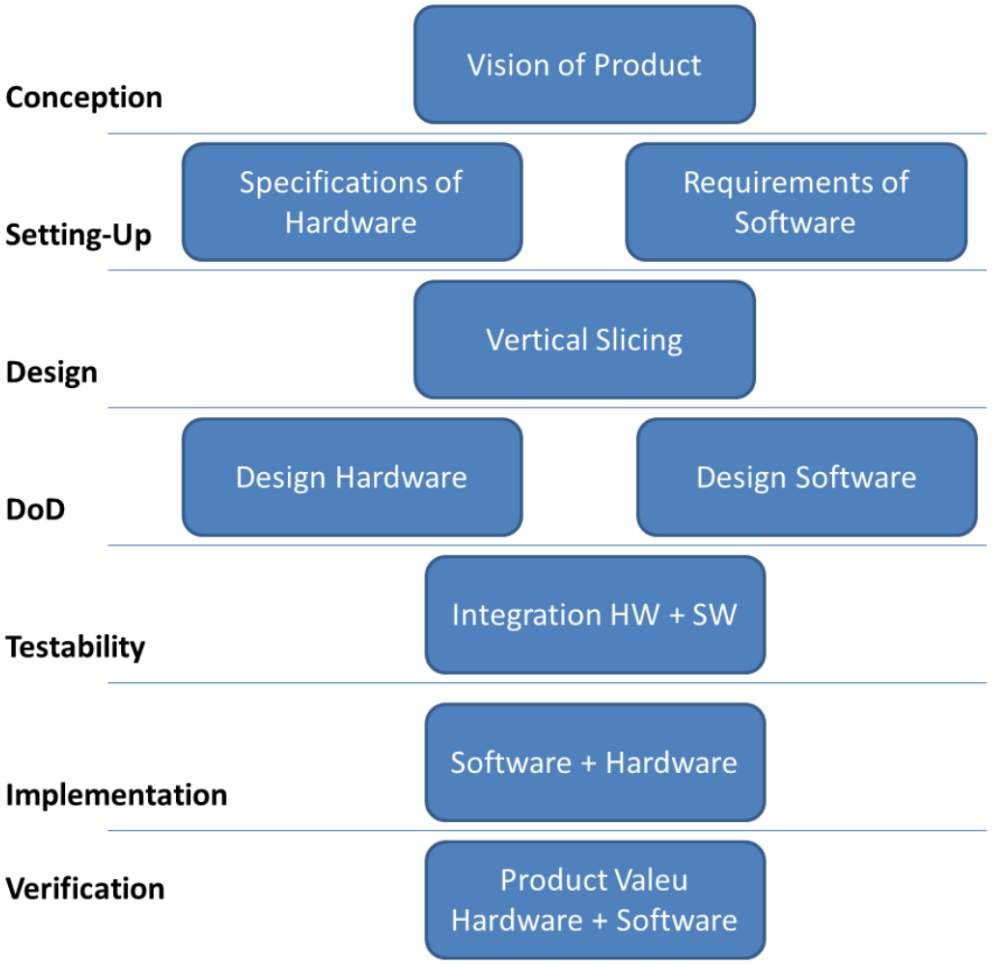
What is Chisel

- A hardware construction language developed by UC Berkely
- Embedded in high-level programming language Scala
- Object-oriented and functional programming
- Register-transfer level description (DFG)
- Convertible to Verilog

Why Chisel

Troubles facing hardware design community

- Hardware Design is slow
- Each iteration is nearly independent
- Most sources are confidential, incompatible
- Cooperation with software becomes common



Why Chisel

A sweet point between traditional hardware design language and pure software programming

Chisel vs. Verilog/VHDL

+ Productive

+ Intuitive

+ Synthesis Optimization

- Control

- Performance

Chisel vs. Xilinx HLS

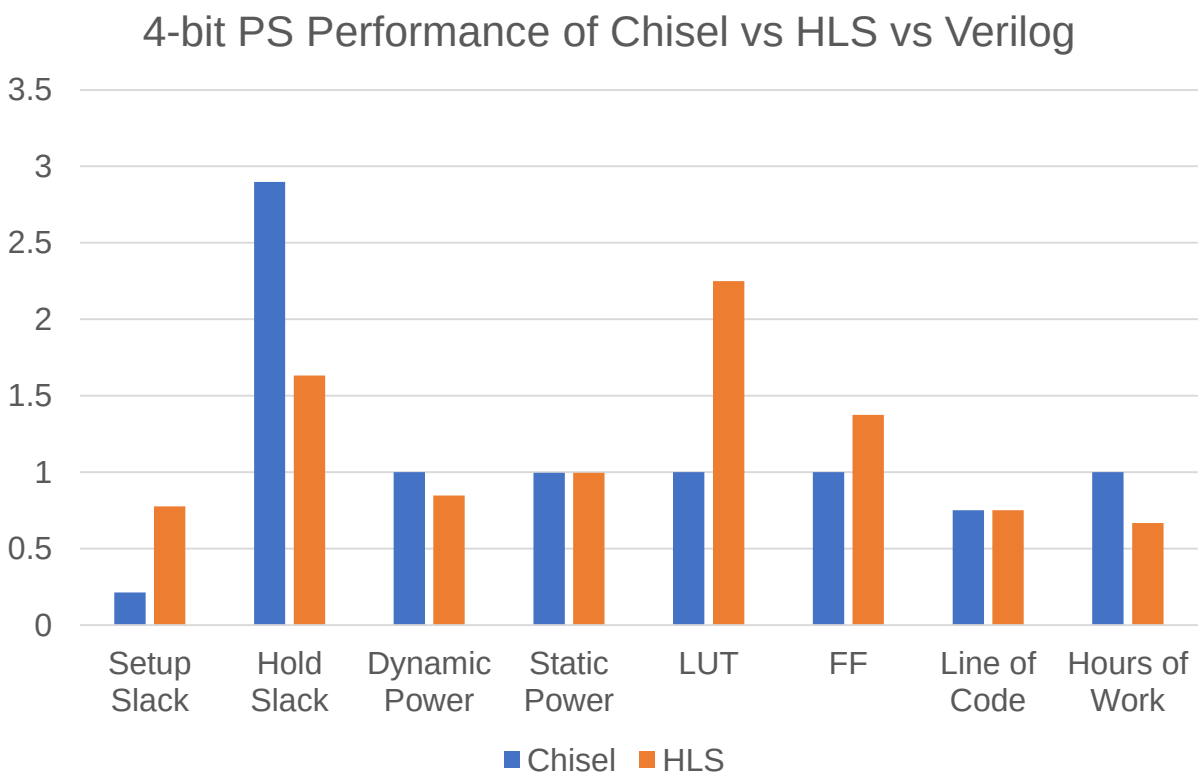
+ Control

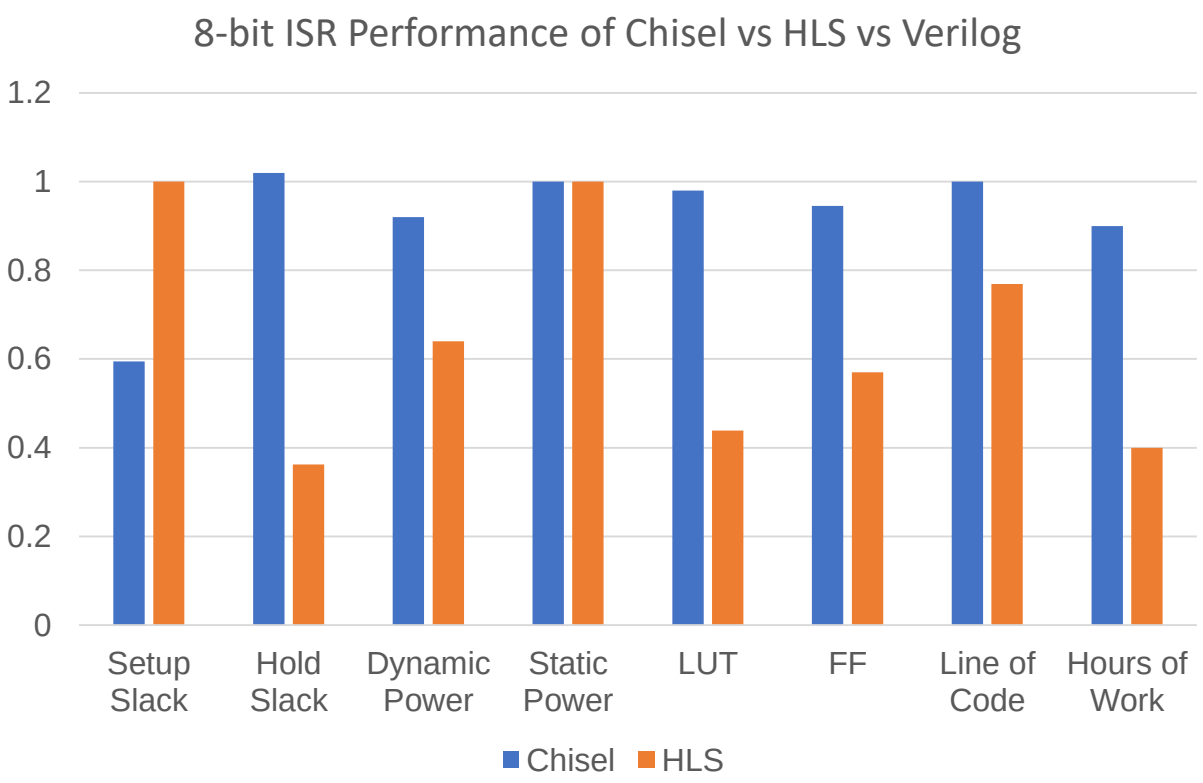
+ Performance

- Productivity

- Synthesis Optimization

Why Chisel


4-bit PS Performance of Chisel vs HLS vs Verilog
3.5
3
2.5
2
1.5
1
0.5
0
Setup Slack
Hold Slack
Dynamic Power
Static Power
LUT
FF
Line of Code
Hours of Work
Chisel
HLS


8-bit ISR Performance of Chisel vs HLS vs Verilog
1.2
1
0.8
0.6
0.4
0.2
0
Setup Slack
Hold Slack
Dynamic Power
Static Power
LUT
FF
Line of Code
Hours of Work
Chisel
HLS

A Chisel Program Looks like

Example: 4-bit Priority Selector

```
class ps4_behavior extends Module {
  val io = IO(new Bundle {
    val input        = Input(UInt(4.W))
    val output     = Output(UInt(4.W))
    val enable       = Input(UInt(1.W))
    val reset       = Input(UInt(1.W))
  })

  val reg_in = RegInit(0.U(4.W))
  val reg_out = RegInit(0.U(4.W))
  val temp_out = Wire(UInt(4.W))

  when (((reg_in >> 3) & 1.U) === 1.U) {
    temp_out := 8.U
  } .elsewhen (((reg_in >> 2) & 1.U) === 1.U) {
    temp_out := 4.U
  } .elsewhen (((reg_in >> 1) & 1.U) === 1.U) {
    temp_out := 2.U
  } .elsewhen ((reg_in & 1.U) === 1.U) {
    temp_out := 1.U
  } otherwise {
    temp_out := 0.U
  }

  when (io.reset === 1.U) {
    reg_in := 0.U
    reg_out := 0.U
  } .elsewhen (io.enable === 1.U) {
    reg_in := io.input
    reg_out := temp_out
  }

  io.output := reg_out;
}
```

Primitive Datatypes

Type	Construction	Constant	Assignment
Unsigned Int	UInt(8.W)	0.U “ha”.U “o12”.U “b1010”.U	val a = UInt(25)
Signed Int	SInt(16.W)	-3.S	val b = SInt(-3)
Boolean (Binary)	Bool()	true.B	val c = Bool(false)

Aggregate Datatypes

Bundle

- Similar to structs
- A group of variables of different types
- Indexed by name

```
class BundleVec extends Bundle {
  val field = UInt(8.W)
  val vector = Vec(4,UInt(8.W))
}
```

Vector

- Similar to array
- A group of variables of same type
- Indexed by number

```
val registerFile = Reg(Vec(32, UInt(32.W)))
```

Combinational Logic

Operator	Description	Data types
`* / %`	multiplication, division, modulus	UInt, SInt
`+ -`	addition, subtraction	UInt, SInt
`=== =/=`	equal, not equal	UInt, SInt, returns Bool
`> >= < <=`	comparison	UInt, SInt, returns Bool
`<< >>`	shift left, shift right (sign extend on SInt)	UInt, SInt
`~`	NOT	UInt, SInt, Bool
`& | ^`	AND, OR, XOR	UInt, SInt, Bool
`!`	logical NOT	Bool
`&& ||`	logical AND, OR	Bool

Function	Description	Data types
`v.andR v.orR v.xorR`	AND, OR, XOR reduction	UInt, SInt, returns Bool
`v(n)`	extraction of a single bit	UInt, SInt
`v(end, start)`	bitfield extraction	UInt, SInt
`Fill(n, v)`	bitstring replication, n times	UInt, SInt
`Cat(a, b, ...)`	bitfield concatenation	UInt, SInt

Register

- Initialize (value on reset)

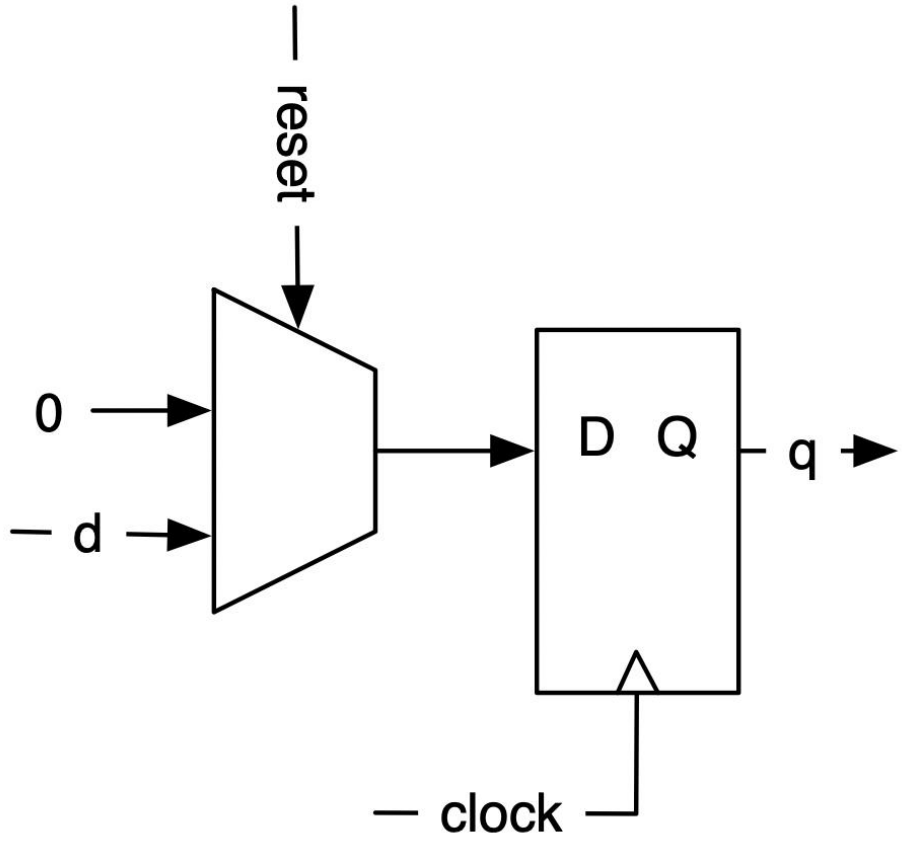
```
val reg = RegInit(0.U(8.W))
```

- Assign (connect input/output)

```
reg := d
val q = reg
```

- Initialize and Input Assign

```
val bothReg = RegNext(d, 0.U)
```



Mux

- If-else
- Switch
- 2-to-1 Mux

  val result = Mux(sel, a, b)

- 4-to-1 Mux

  val result = Mux(sel(1), a, Mux(sel(2), b, Mux(sel(3), c, d)),

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

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

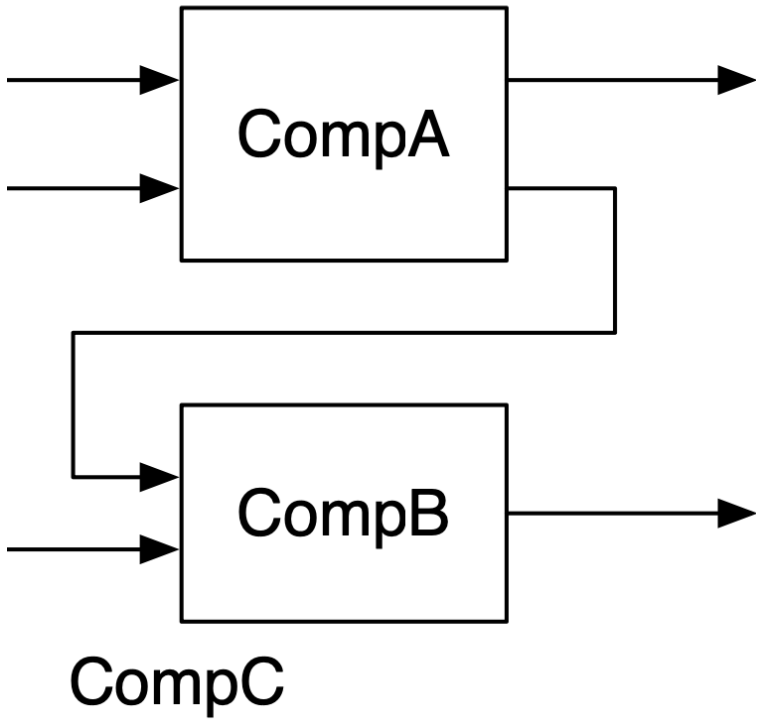
```
switch(sel) {
  is (0.U) { result := 1.U}
  is (1.U) { result := 2.U}
  is (2.U) { result := 4.U}
  is (3.U) { result := 8.U}
}
```

Module

```
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
  io.out_y := compB.io.out
}
```



PeekPoke Test

Example: 4-bit Priority Selector

```
class ps4_behavior extends Module {
  val io = IO(new Bundle {
    val input        = Input(UInt(4.W))
    val output     = Output(UInt(4.W))
    val enable       = Input(UInt(1.W))
    val reset       = Input(UInt(1.W))
  })

  val reg_in = RegInit(0.U(4.W))
  val reg_out = RegInit(0.U(4.W))
  val temp_out = Wire(UInt(4.W))

  when (((reg_in >> 3) & 1.U) === 1.U) {
    temp_out := 8.U
  } .elsewhen (((reg_in >> 2) & 1.U) === 1.U) {
    temp_out := 4.U
  } .elsewhen (((reg_in >> 1) & 1.U) === 1.U) {
    temp_out := 2.U
  } .elsewhen ((reg_in & 1.U) === 1.U) {
    temp_out := 1.U
  } otherwise {
    temp_out := 0.U
  }

  when (io.reset === 1.U) {
    reg_in := 0.U
    reg_out := 0.U
  } .elsewhen (io.enable === 1.U) {
    reg_in := io.input
    reg_out := temp_out
  }

  io.output := reg_out;
}
```

PeekPoke Test

Poke

- Set stimuli/input

Poke

- Monitor output

Step

- Wait some number of cycles

```
class TesterBehavior(dut: ps4_behavior) extends PeekPokeTester(dut) {

  poke(dut.io.input, 5.U)
  poke(dut.io.enable, 1.U)
  poke(dut.io.reset, 0.U)
  step(1)
  step(1)
  println("Result is: " + peek(dut.io.output).toString)

}
object  TesterBehavior extends App {
  chisel3.iotesters.Driver(() => new ps4_behavior()) { c =>
    new  TesterBehavior(c)
  }
}
```

Function

- A lightweight alternative of module
- Helpful when logic is simple
- Can be fancy as well

```
def adder (x: UInt, y: UInt) = {
  x + y
}
```

```
val x = adder(a, b)
// another adder
val y = adder(c, d)
```

Function

- Pulse-width modulation

```
def pwm(nrCycles: Int, din: UInt) = {
  val cntReg = RegInit(0.U(unsignedBitLength(nrCycles-1).W))
  cntReg := Mux(cntReg === (nrCycles-1).U, 0.U, cntReg + 1.U)
  din > cntReg
}

val din = 3.U
val dout = pwm(10, din)
```

Parameterization

- Data parameter
- Scala type (not Chisel primitive)
- Conversion may be needed

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val c = Output(UInt(n.W))
  })

  io.c := io.a + io.b
}

val add8 = Module(new ParamAdder(8))
val add16 = Module(new ParamAdder(16))
```

Parameterization

- Type parameter
- Works for primitive & aggregate

```
class NocRouter[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val address = Input(Vec(n, UInt(8.W)))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...

  class Payload extends Bundle {
    val data = UInt(16.W)
    val flag = Bool()
  }


  val router = Module(new NocRouter(new Payload, 2))
```

Inheritance

- A parent class with multiple variances
- Code reuse, only differs at declaration

```
abstract class Ticker(n: Int) extends Module {
  val io = IO(new Bundle{
    val tick = Output(Bool())
  })
}

class TickerTester[T <: Ticker](dut: T, n: Int) extends
    PeekPokeTester(dut: T) {

  // -1 is the notion that we have not yet seen the first tick
  var count = -1
  for (i <- 0 to n * 3) {
    if (count > 0) {
      expect(dut.io.tick, 0)
    }
    if (count == 0) {
      expect(dut.io.tick, 1)
    }
    val t = peek(dut.io.tick)
    // On a tick we reset the tester counter to N-1,
    // otherwise we decrement the tester counter
    if (t == 1) {
      count = n-1
    } else {
      count -= 1
    }

    step(1)
  }
}
```

Inheritance

```
class UpTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

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

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

  io.tick := cntReg === N
}
```

```
class DownTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(N)

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

  io.tick := cntReg === N
}
```

Inheritance

```
"UpTicker 5" should "pass" in {
  chisel3.iotesters.Driver(() => new UpTicker(5)) { c =>
    new TickerTester(c, 5)
  } should be (true)
}

"DownTicker 7" should "pass" in {
  chisel3.iotesters.Driver(() => new DownTicker(7)) { c =>
    new TickerTester(c, 7)
  } should be (true)
}
```

Resources

- Digital Design with Chisel (the one I use)

http://www.imm.dtu.dk/~masca/chisel-book.pdf

- Install Chisel & Code Example

https://github.com/ucb-bar/chisel-tutorial

- Berkely Short Tutorial

https://inst.eecs.berkeley.edu/~cs250/sp17/handouts/chisel-tutorial.pdf

- Berkely RISC-V Bootcamp

https://www.youtube.com/watch?v=pfM1WUWbfBs&ab_channel=RISC-VInternational

https://riscv.org/wp-content/uploads/2015/01/riscv-chisel-tutorial-bootcamp-jan2015.pdf

Design flow

- Understand Algorithm
- Breakdown components
- Determine Interface
- Draw DFG
- Map DFG to Chisel code
- Test and optimize

```
// Problem:
//
// This module should be able to write 'data' to
// internal memory at 'wrAddr' if 'isWr' is asserted.
//
// This module should perform sequential search of 'data'
// in internal memory if 'en' was asserted at least 1
clock cycle
//
// While searching 'done' should remain 0,
// 'done' should be asserted when search is complete
//
// If 'data' has been found 'target' should be updated to
the
// address of the first occurrence
//
```

Demo