

Chisel: Memory



Muhammad Tahir*

Lecture 8

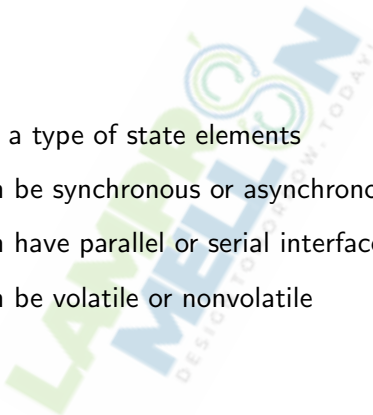
Contents

- 1 Chisel Memory Constructors
- 2 Memory Parameterization
- 3 Code Memory
- 4 Data Memory
- 5 Memory Forwarding



Memory

- Memories are a type of state elements
- Memories can be synchronous or asynchronous
- Memories can have parallel or serial interface
- Memories can be volatile or nonvolatile



Chisel Memory Constructor: Mem

Chisel supports two memory constructors

Mem: Asynchronous read and synchronous write memory

- Supports *read* and *write* methods
- Write is sequential and takes effect on the rising clock edge after the write request
- Masked write is supported
- Read operation is combinational
- In case of multiple conflicting writes, the result is undefined
- Read-after-write hazard is not an issue

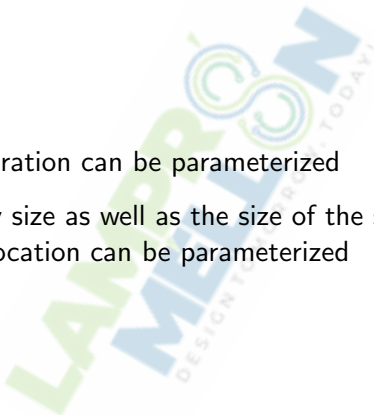
Chisel Memory Constructor: `SyncReadMem`

`SyncReadMem`: A synchronous-read, synchronous-write memory

- Supports *read* and *write* methods
- Writes take effect on the rising edge
- Masked write is supported
- Read returns data on the clock rising edge after the request
- Read-after-write behavior (when a read and write to the same address are requested on the same cycle) is undefined

Memory Parameterization

- Memory generation can be parameterized
- Both memory size as well as the size of the smallest addressable location can be parameterized



Memory Parameterization Cont'd

```
// parameterized memory
import chisel3._
import chisel3.util._

class Parameterized_Mem(val size: Int = 32, val width: Int = 32)
  extends Module {
  val io = IO(new Bundle {
    val dataIn = Input(UInt(width.W))
    val dataOut = Output(UInt(width.W))
    val addr = Input(UInt(log2Ceil(size).W))
    val rd_enable = Input(Bool())
    val wr_enable = Input(Bool())
  })

  val Sync_memory = SyncReadMem(size, UInt(width.W))
  // memory write operation
  when(io.wr_enable){
    Sync_memory.write(io.addr, io.dataIn)
  }
  io.dataOut := Sync_memory.read(io.addr, io.rd_enable)
}

println((new chisel3.stage.ChiselStage).emitVerilog(new
  Parameterized_Mem))
```

Implementing Register File

```
import chisel3._

class RegFileIO extends Bundle with Config {
  val raddr1 = Input(UInt(5.W))
  val raddr2 = Input(UInt(5.W))
  val rdata1 = Output(UInt(XLEN.W))
  val rdata2 = Output(UInt(XLEN.W))
  val wen    = Input(Bool())
  val waddr  = Input(UInt(5.W))
  val wdata  = Input(UInt(XLEN.W))
}

class RegFile extends Module with Config {
  val io = IO(new RegFileIO)
  val regs = Mem(REGFILE_LEN, UInt(XLEN.W))

  io.rdata1 := Mux((io.raddr1.orR), regs(io.raddr1), 0.U)
  io.rdata2 := Mux((io.raddr2.orR), regs(io.raddr2), 0.U)

  when(io.wen & io.waddr.orR) {
    regs(io.waddr) := io.wdata
  }
}
```


Code Memory with Initialization

```
package LM_Chisel

import chisel3._
import chisel3.util._
import chisel3.util.experimental.loadMemoryFromFile
import scala.io.Source

class InstMemIO extends Bundle with Config {
  val addr = Input(UInt(WLEN.W))
  val inst = Output(UInt(WLEN.W))
}

class InstMem(initFile: String) extends Module with Config {
  val io = IO(new InstMemIO)

  // INST_MEM_LEN in Bytes or INST_MEM_LEN / 4 in words
  val imem = Mem(INST_MEM_LEN, UInt(WLEN.W))

  loadMemoryFromFile(imem, initFile)

  io.inst := imem(io.addr / 4.U)
}
```

Code Memory with Initialization Cont'd

- File to be loaded to instruction memory is passed as string parameter by the top module

```
object Generate_ProcessorTile extends App {  
  var initFile = "src/test/resources/main.txt"  
  
  chisel3.Driver.execute(args, () => new ProcessorTile(  
    initFile))  
}
```

- A separate .v file is generated for binding instruction memory to the executable file

Data Memory

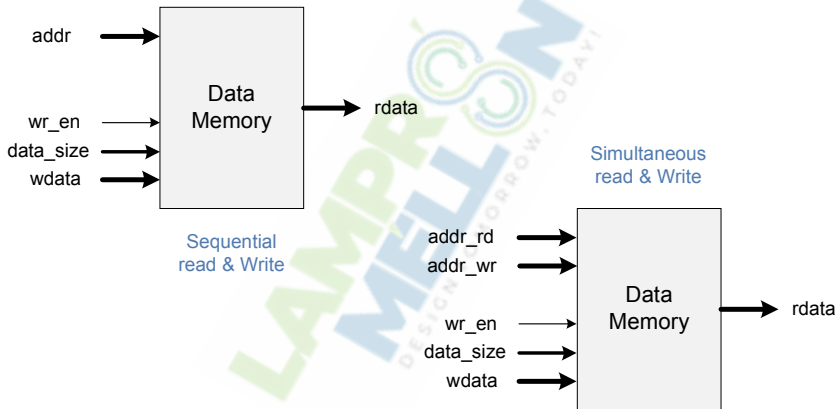


Figure: Data memory interfaces.

Data Memory

```
package LM_Chisel

import chisel3._
import chisel3.util._
import Control._

class DataMemIO extends Bundle with Config {
  val addr      = Input(UInt(WLEN.W))
  val wdata     = Input(UInt(WLEN.W))
  val rd_en     = Input(Bool())
  val wr_en     = Input(Bool())
  val st_type   = Input(UInt(STTYPE_LEN.W))
  val rdata     = Output(UInt(WLEN.W))
}

class DataMem extends Module with Config {
  val io = IO(new DataMemIO)

  // Data memory size and addressability width
  val dmem = SyncReadMem(DATA_MEM_LEN, UInt(BLEN.W))
  val addr = io.addr
  val read_data = Wire(UInt(XLEN.W))
  read_data := 0.U
}
```

Data Memory Cont'd

```
when (io.wr_en) {  
  when (io.st_type === 1.U) {  
    dmem (addr) := io.wdata(7,0)  
    dmem (addr + 1.U) := io.wdata(15,8)  
    dmem (addr + 2.U) := io.wdata(23,16)  
    dmem (addr + 3.U) := io.wdata(31,24)  
  }.elsewhen (io.st_type === 2.U) {  
    dmem (addr) := io.wdata(7,0)  
    dmem (addr + 1.U) := io.wdata(15,8)  
  }.elsewhen (io.st_type === 3.U) {  
    dmem (addr) := io.wdata(7,0)  
  }  
}  
  
// read data from 4 memory banks  
read_data := Cat(dmem(addr + 3.U), dmem(addr + 2.U), dmem(addr +  
  1.U), dmem(addr))  
  
io.rdata := Mux(io.rd_en, read_data, 0.U)  
}
```

Data Memory with Masking

```
package LM_Chisel

import chisel3._
import chisel3.util._

class DataMemIO extends Bundle with Config {
  val addr      = Input(UInt(WLEN.W))
  val wdata     = Input(UInt(WLEN.W))
  val rd_en     = Input(Bool())
  val wr_en     = Input(Bool())
  val mask      = Input(Vec(4, Bool()))
  val rdata     = Output(UInt(WLEN.W))
}

class DataMem extends Module with Config {
  val io = IO(new DataMemIO)

  // Data memory size and addressability width
  val dmem = SyncReadMem(DATA_MEM_LEN, UInt(BLEN.W))
  // Write with mask
  when (io.wr_en.toBool()) {
    dmem.write(io.addr, io.wdata, io.mask)
  }
  io.rdata := dmem.read(io.addr, rd_en)
}
```

Memory Forwarding

- When do we need memory forwarding?
- Memory read and write operations during the same clock cycle addressing same memory location
- Problem becomes more complex in case of multi-port memories
- What about register files?
- How do we resolve this issue for register files?

Memory Forwarding

Forwarding in memory can be implemented as shown

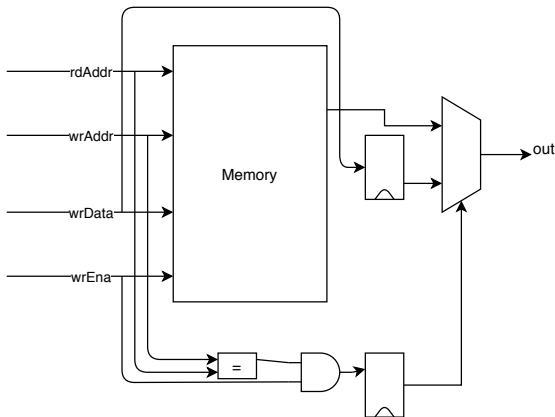


Figure: Memory forwarding implementation.

Memory Forwarding Cont'd

```
// Memory forwarding example
import chisel3._
import chisel3.util._

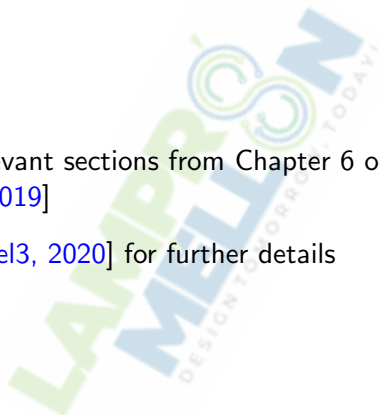
class Forwarding extends Module {
  val io = IO(new Bundle{
    val out = Output(UInt(32.W))
    val rdAddr = Input(UInt(10.W))
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(32.W))
    val wr_en = Input(Bool())
  })

  val memory = SyncReadMem(1024, UInt(32.W))
  val wrDataReg = RegNext(io.wrData)
  val doForwardReg = RegNext(io.wrAddr === io.rdAddr && io.wr_en)
  val memData = memory.read(io.rdAddr)
  when(io.wr_en) {
    {
      memory.write(io.wrAddr, io.wrData)
    }
  }
  io.out := Mux(doForwardReg, wrDataReg, memData)
}

println((new chisel3.stage.ChiselStage).emitVerilog(new Forwarding()))
```

Reading List I

- Read the relevant sections from Chapter 6 of [[Schoeberl, 2019](#)]
- Consult [[chisel3, 2020](#)] for further details



References



chisel3 (2020).

Chisel3 library reference.

<https://www.chisel-lang.org>.



Schoeberl, M. (2019).

Digital Design with Chisel.

Kindle Direct Publishing.

