Experiment 8

Memories

Objective

Understand different memory constructs available in Chisel and learn their use for instruction and data storage.

8.1 Memory in Chisel

Memories can be synchronous as well as asynchronous. In addition, the memories can have multiple ports. In Chisel, we can define the following two types of memories with single port memory interface.

- Asynchronous read, synchronous write
- Synchronous read and synchronous write

8.1.1 Asynchronous Memory

Asynchronous memories can be made in *Chisel* using `mem` construct. We have already seen asynchronous memory in Lab 2. There are two methods associated with `mem`.

- **Write:** To perform write operation we call method *write* on the `mem` object and provide data and address address as arguments. The data is written to the specified address on the positive edge of the clock.
- **Read:** The Read method defined on `mem` object is combinational, which implies data is available immediately after applying the address. Depending on the memory model used for `mem` implementation, immediately, can imply either no delay or some constant delay corresponding to combinational implementation of address decoding with some propagation delay offset. When performing a read operation, we only need to provide address as an argument.

The implementation of asynchronous memory using `mem` and use of read and write methods is illustrated in Listing 8.1

```
// mem Example
import chisel3._

class IO_Interface extends Bundle{
  // Make an input from a Vector of 4 values
  val data_in = Input(Vec(4,(UInt(32.W))))

  // Signal to control which vector is selected
```

```
    val data_selector = Input(UInt(2.W))
    val data_out = Output(UInt(32.W))
    val addr = Input(UInt(5.W))

    // The signal is high for write
    val wr_en = Input(Bool())
}

class Asynch_Mem extends Module {
    val io = IO(new IO_Interface)

    io.data_out := 0.U

    // Make a memory of 32x32
    val memory = Mem(32, UInt(32.W))

    when(io.wr_en){
        // Write for wr_en = 1
        // Write at memory location addr, with selected data from data_in
        memory.write(io.addr, io.data_in(io.data_selector))
    }
    // Asyncronous read from addr location
    io.data_out := memory.read(io.addr)
}

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

Listing 8.1: Asynchronous memory using `mem`.

8.1.2 Synchronous

Synchronous memories have read operation also synchronized to the clock. We can make synchronous memories in *Chisel* using `SyncReadMem`. When instantiating memory using `SyncReadMem`, we need to specify memory size in terms of number of locations as well as data-type and width of each location. For instance, we can implement word addressable synchronous memory with 1024 32-bit locations, using the following syntax.

```
val memory = SyncReadMem (1024, UInt(32.W))
```

8.1.3 Memory Parameterization

Memory generation can be parameterized by specifying memory size as well as the size of the smallest addressable location as parameters. Listing 8.2 provides an illustration to implement parameterized memory.

```
// 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))
```

Listing 8.2: Parameterized memory using `SyncReadMem`.

8.1.4 Implementing Register File

We have implemented a register file using `Vec` in Lab 6. We can also implement the register file using memory. This can be easily achieved by replacing the register vector with memory as illustrated in the following syntax.

```
val regFile = Mem (32, UInt(32.W))
```

8.1.5 Initializing Code memory

Code memory can be implemented using either synchronous or asynchronous memories. When testing the functionality of a processor, one needs to load the code memory with the executable of user program. When simulation is the mode of testing, then one needs to initialize the code memory with the user program. This can be achieved by using `loadMemoryFromFile` from Chisel library utility. Listing 8.3 illustrates this functionality.

```
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)
}
```

Listing 8.3: Initializing code memrory.

User executable file, to be loaded to instruction memory, is passed as string parameter by the top module as illustrated in Listing 8.4. A separate .v file is generated during hardware generation that is used for binding instruction memory to executable file.

```
object Generate_ProcessorTile extends App {
  var initFile = "src/test/resources/main.txt"

  chisel3.Driver.execute(args, () => new ProcessorTile(initFile))
}
```

Listing 8.4: Top module passing user executable file as paramater.

8.1.6 Memory with Forwarding

Forwarding is required when write and read operations are performed on the same memory location during the same cycle. This can happen with memories supporting simultaneous read and write operations [?]. Contrary to this, memory with sequential read and write operations does not encounter this problem. Memory interfaces using sequential and simultaneous read-write operations are depicted in Figure 8.1.

In forwarding, the read operation returns the new write value instead of the previously stored value. To achieve compatibility with the expected one-cycle read-after-write latency behavior, the write address and data are put through register to delay by one cycle. Block diagram illustrating forwarding implementation is shown in Figure 8.2. The implementation of forwarding logic is given in Listing 8.5.

```
// 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))
```

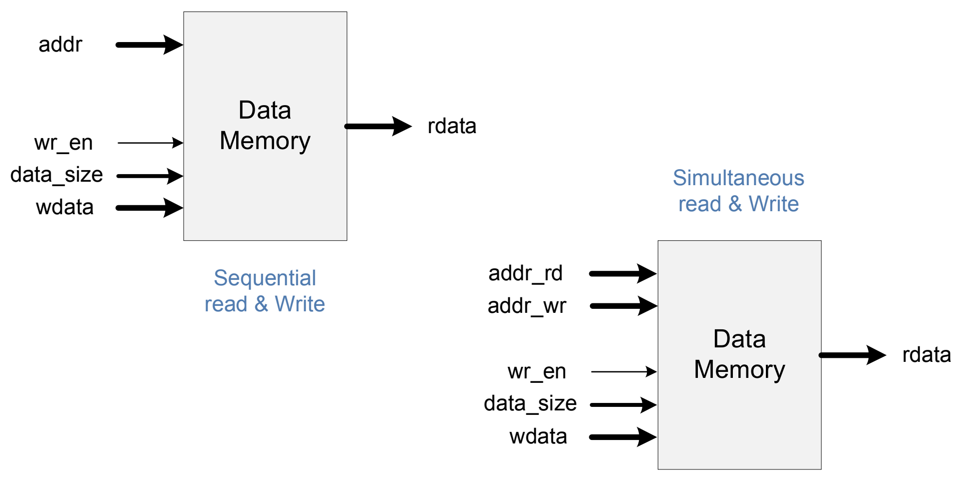

Figure 8.1: Data memory interfaces.

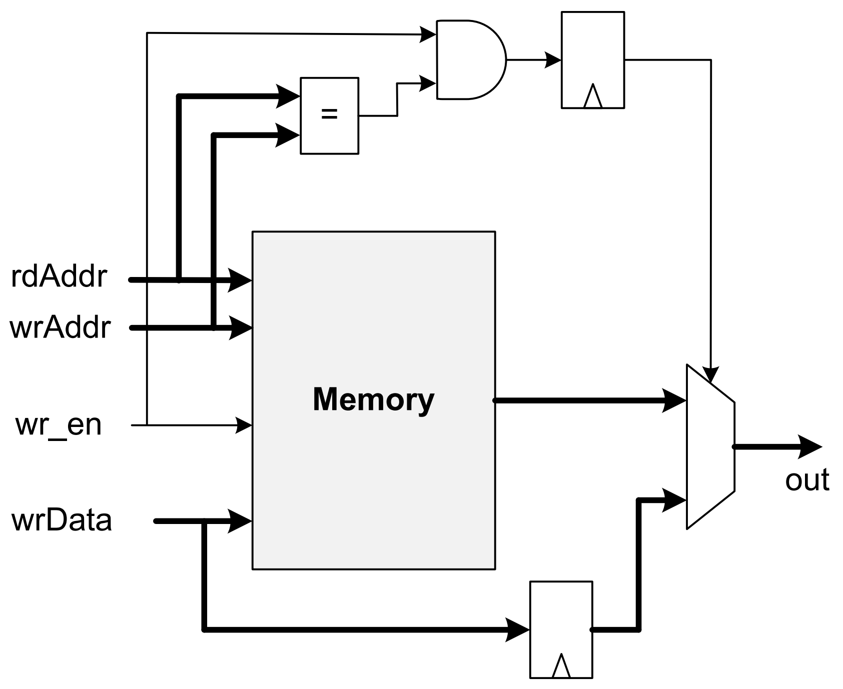

Figure 8.2: Memory forwarding.

```
    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()))
```

Listing 8.5: Memory forwarding using `mem`.

8.1.7 Memory with Mask

Previously, we have used write method with `SyncReadMem` using two arguments, that is address and data. However, the write method also supports a third argument, which is the *mask*. Masking is required when performing byte and half-word access from a word addressable location. To illustrate the use of mask we define memory with 32-bit width and can be byte masked.Implementation of such a memory is illustrated in Listing 8.6.

```
import chisel3._
class MaskedReadWriteSmem extends Module {
   val width: Int = 8
   val io = IO(new Bundle {
      val enable = Input(Bool())
      val write = Input(Bool())
      val addr = Input(UInt(10.W))
      val mask = Input(Vec(4, Bool()))
      val dataIn = Input(Vec(4, UInt(width.W)))
      val dataOut = Output(Vec(4, UInt(width.W)))
   })

   // Create a 32-bit wide memory that is byte-masked
   val mem = SyncReadMem(1024, Vec(4, UInt(width.W)))
   // Write with mask
   mem.write(io.addr, io.dataIn, io.mask)
   io.dataOut := mem.read(io.addr, io.enable)
}

println(chisel3.Driver.emitVerilog(new MaskedReadWriteSmem))
```

Listing 8.6: Memory with mask.

8.2 Exercises

Exercise 1: Try to write code in Listing 8.6 without using overloaded method for masking.

Exercise 2: Make forwarding for a 2-banked memory, refer to Listing 8.6.

Exercise 3: Write instruction steam to a memory and verify it.

8.3 Assignments

Task 1: Make memory bank that is accessed by an arbiter. The arbiter have 4 requestors, each is a queue. The arbiter selects a queue and writes the data to bank memory. Bank is selected on accordance of the requestor selected. For example if requestor 2 was select then the data will be written to the bank 2 and vice versa. A block diagram of the assignment is shown below in Figure 8.3. A skeleton code for the assignment is provided in Listing 8.7.

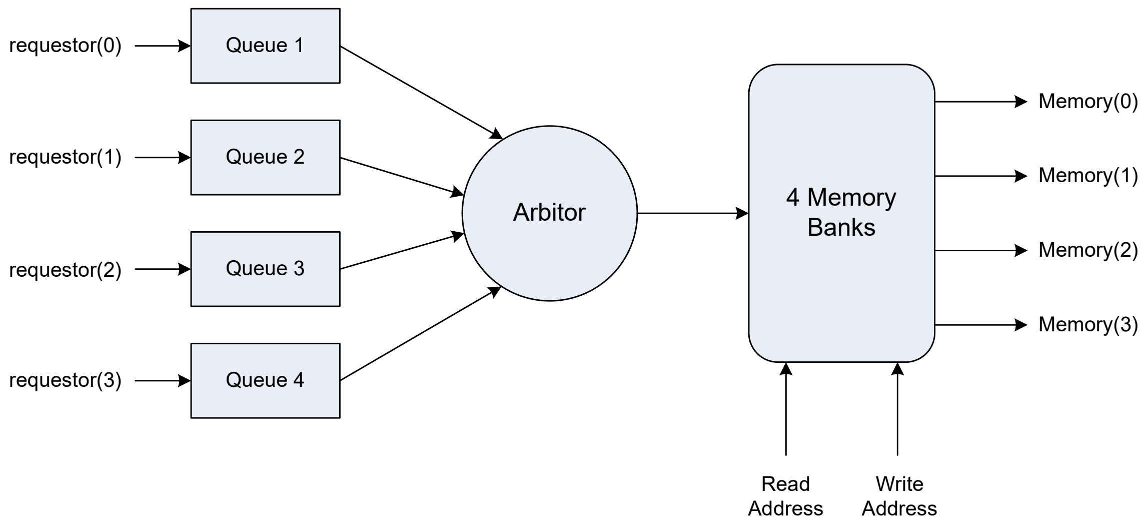

Figure 8.3: Memory assignment.

```
package Lab8

import chisel3._
import chisel3.util._

class memory_assignment extends Module {
   val io = IO(new Bundle{
      val memory_out = Vec(4, Output(UInt(32.W)))
      val requestor = Vec(4, Flipped(Decoupled(UInt(32.W))))
      val Readaddr = Input(UInt(5.W))
      val Writeaddr = Input(UInt(5.W))
   })
   // Start your code from here

   // End your code here
}
```

Listing 8.7: Skeleton code for the assignment