A2. Logic Design in Chisel

CSE261: Computer Architecture, Fall 2021

Hyungon Moon

Out: Oct 26, 2021

Due: Nov 4 2021 11:59pm

Goal

- Be familiar with Logic Design in Chisel
- Implement some basic elements

Vagrant and VirtualBox

- Vagrant allows you to easily create and access a virtual machine.
- Install Vagrant from

```
https://www.vagrantup.com/
```

- Vagrant (in this class) need VirtualBox. Install from

```
https://www.virtualbox.org/
```

- Should work on Ubuntu, Mac or Windows.
- May not work inside another virtual machine.

Initializing VM (Mac, Ubuntu (Linux))

- Use terminal to enter the `hw1` directory
- Make sure that you are seeing `Vagrantfile` in the directory
- Create and boot VM

```
vagrant up
```

- Connect to the VM

```
vagrant ssh
```

- From V2 of the assignment, the `hw1` directory is automatically synced with the `hw` directory

Initializing VM (Windows)

- Use terminal to enter the `hw1` directory
- Make sure that you are seeing `Vagrantfile` in the directory
- Create and boot VM

```
vagrant.exe up
```

- Connect to the VM

```
vagrant.exe ssh
```

- From V2 of the assignment, the `hw1` directory is automatically synced with the `hw` directory

SBT

- We will use a complex set of tools enabling us to write in Chisel.
- We don't need to care about them: SBT does the dirty job.
  - Located in `hw/sbt` or `sbt`.
- You will use three commands, referring to the lecture slides.
  - To run the Main object,

```
sbt/bin/sbt run
```

  - To run all tests,

```
sbt/bin/sbt test
```

  - To run one test only (e.g., ALUTest)

```
sbt/bin/sbt "testOnly ALUTest"
```

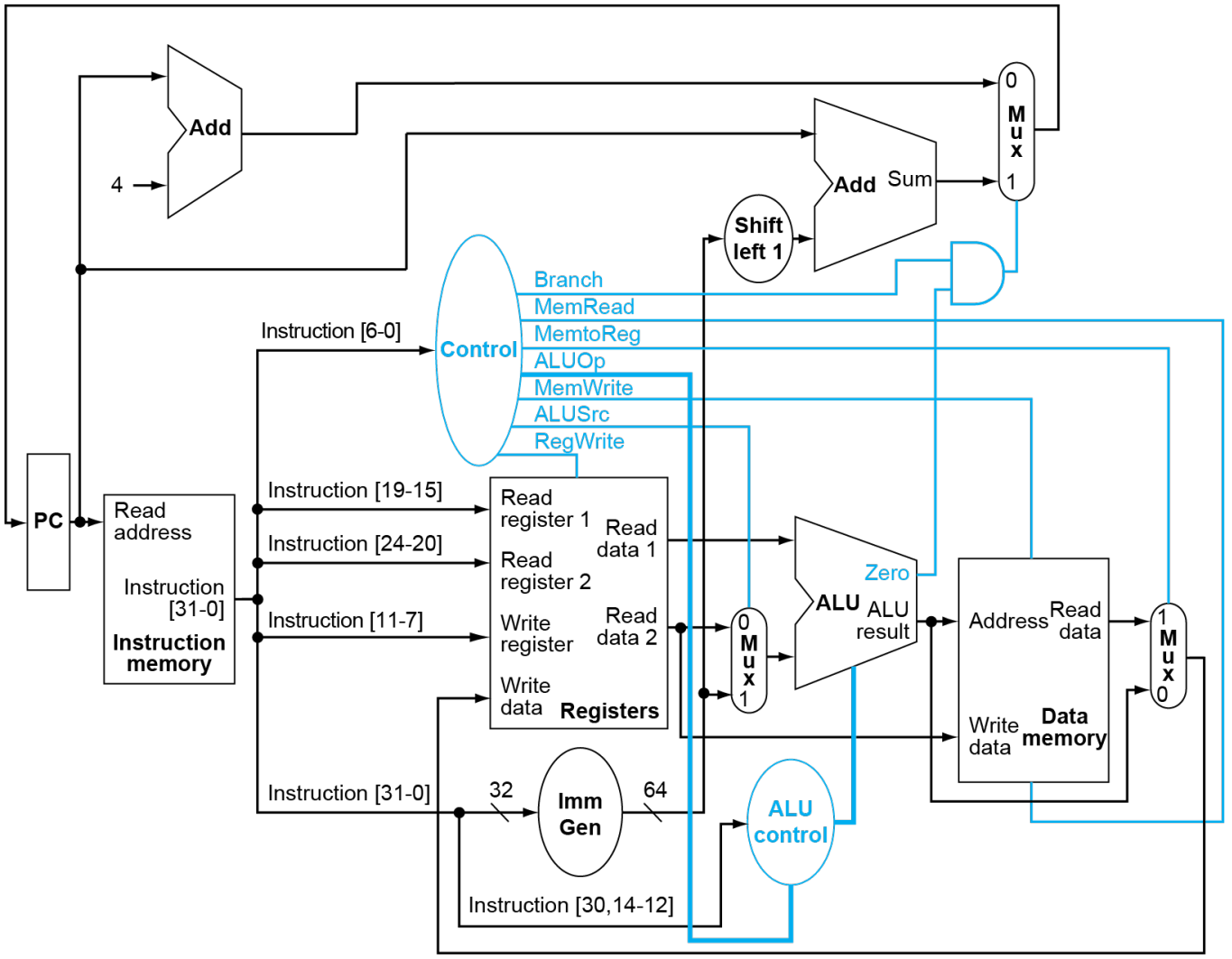
SBT trouble shooting

- When running sbt for the first time, you may see this message

```
java.io.IOException: org.scalasbt.ipcsocket.NativeErrorException:
[1] Operation not permitted
Create a new server? y/n (default y)
```

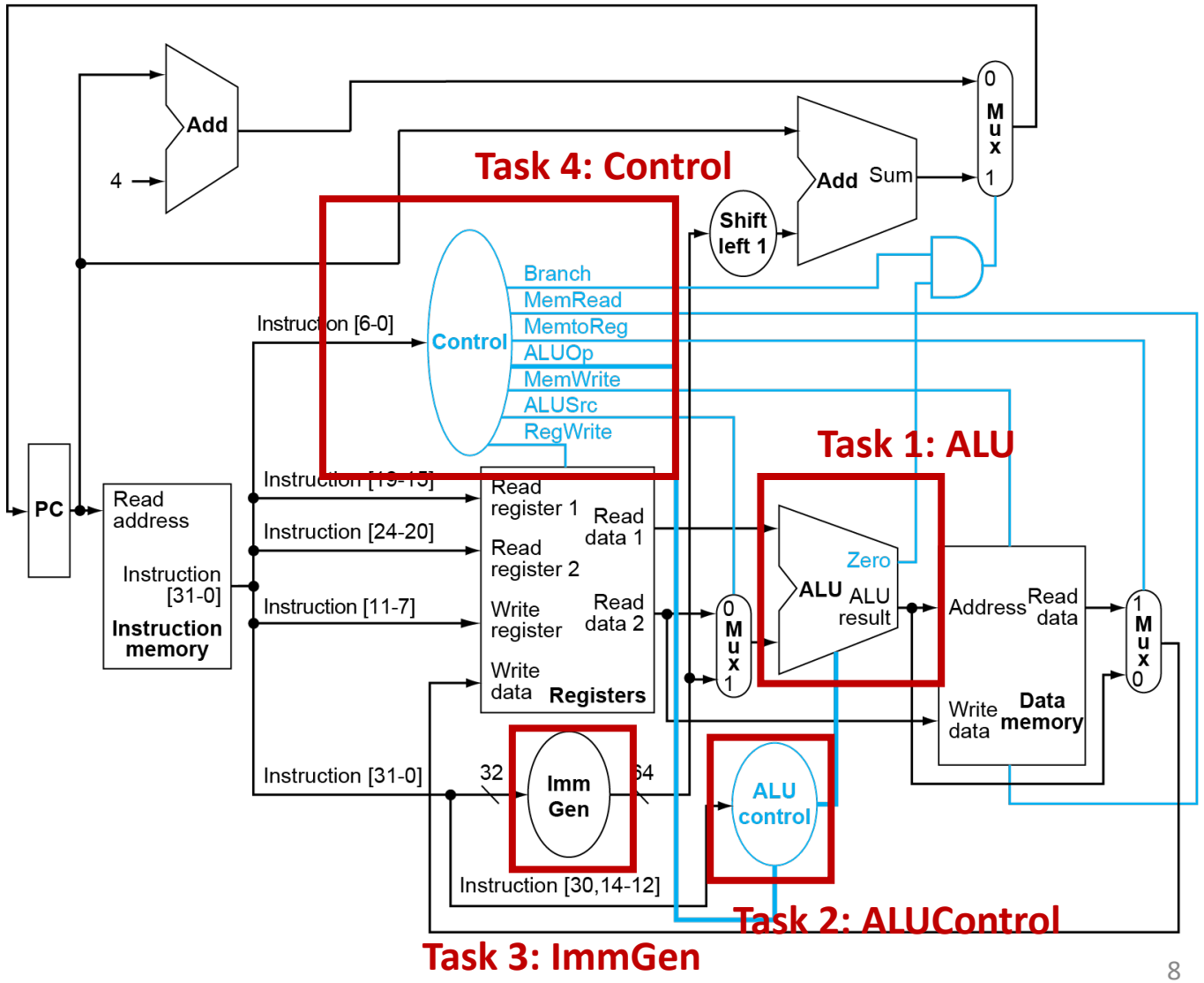
- Just press enter to go for the default value.

Grand goal: implementing the processor



HW2: Some combinational logics

- Execute
  - Load: `ld`
  - Store: `sd`
  - Branch: `beq`
  - Register ops
    - `addi, add, sub, and, or`



Task 0: Passthrough (1/2)

- An example module (also appeared in Lecture)

```
// Chisel Code: Declare a new module
definition
class Passthrough extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(4.W))
    val out = Output(UInt(4.W))
  })
  io.out := io.in
}
```

Task 0: Passthrough (2/2)

- Output example from test

```
sbt/bin/sbt "testOnly PassthroughTest"
```

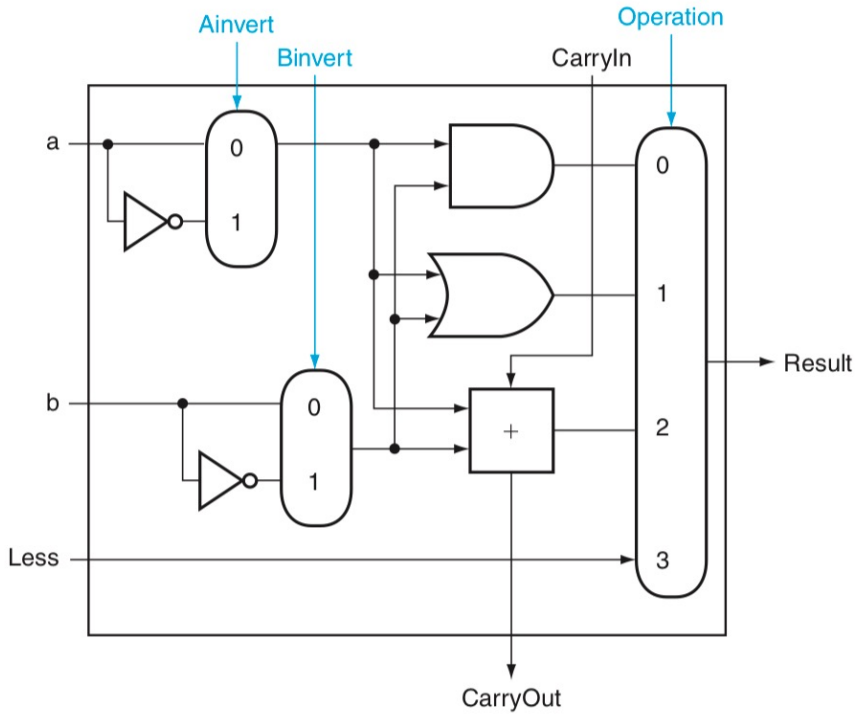
```
test Passthrough Success: 3 tests passed in 6 cycles taking 0.013428 seconds

[info] [0.006] RAN 1 CYCLES PASSED
[info] PassThroughTest:
[info] Passthrough
[info] - should pass the values
[info] ScalaTest
[info] Run completed in 1 second, 393 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```

Task 1: Arithmetic Logic Unit (ALU) (1/3)

- Implement an ALU that implements computations that we need.

```
class ALU extends Module {
  val io = IO(new Bundle {
    val ctrl = Input(UInt(4.W))
    val a = Input(UInt(64.W))
    val b = Input(UInt(64.W))
    val res = Output(UInt(64.W))
    val zero = Output(Bool())
  })
  /* Your code starts here*/
  io.res := 0.U
  io.zero := true.B
  /*Your code ends here */
}
```



Task 1: Arithmetic Logic Unit (ALU) (2/3)

- Should support the five instructions.
- Refer to the following truth table.

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

- `io.zero` should emit `true.B` only if the result of subtraction is zero (for branches)

Task 1: Arithmetic Logic Unit (ALU) (3/3)

- Example output

```
sbt/bin/sbt "testOnly ALUTest"
```

```
test ALU Success: 82 tests passed in 5 cycles taking 0.064239 seconds

[info] [0.054] RAN 0 CYCLES PASSED
[info] ALUTest:
[info] ALU
[info] - should compute
[info] ScalaTest
[info] Run completed in 1 second, 437 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```

Task 2: ALUControl (1/2)

- ALUControl generates `ALUCtrl` from `ALUOp` and instruction.
- Truth table

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
`ld`	00	load register	XXXXXXXXXXX	add	0010
`sd`	00	store register	XXXXXXXXXXX	add	0010
`beq`	01	branch on equal	XXXXXXXXXXX	subtract and compare	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

Task 2: ALUControl (2/2)

- Example Output

```
sbt/bin/sbt "testOnly ALUControlTest"
```

```
test ALUControl Success: 4 tests passed in 5 cycles taking 0.013563 seconds

[info] [0.006] RAN 0 CYCLES PASSED
[info] ALUControlTest:
[info] ImmGen
[info] - should
[info] ScalaTest
[info] Run completed in 1 second, 427 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```

Task 3: ImmGen (1/2)

- Referring to the Lecture slides, implement the ImmGen so that it correctly handles immediate from the instructions
  - Load: `ld`
  - Store: `sd`
  - Branch: `beq`
  - Register ops
    - `addi, add, sub, and, or`

Task 3: ImmGen (2/2)

- Example Output

```
sbt/bin/sbt "testOnly ImmGenTest"
```

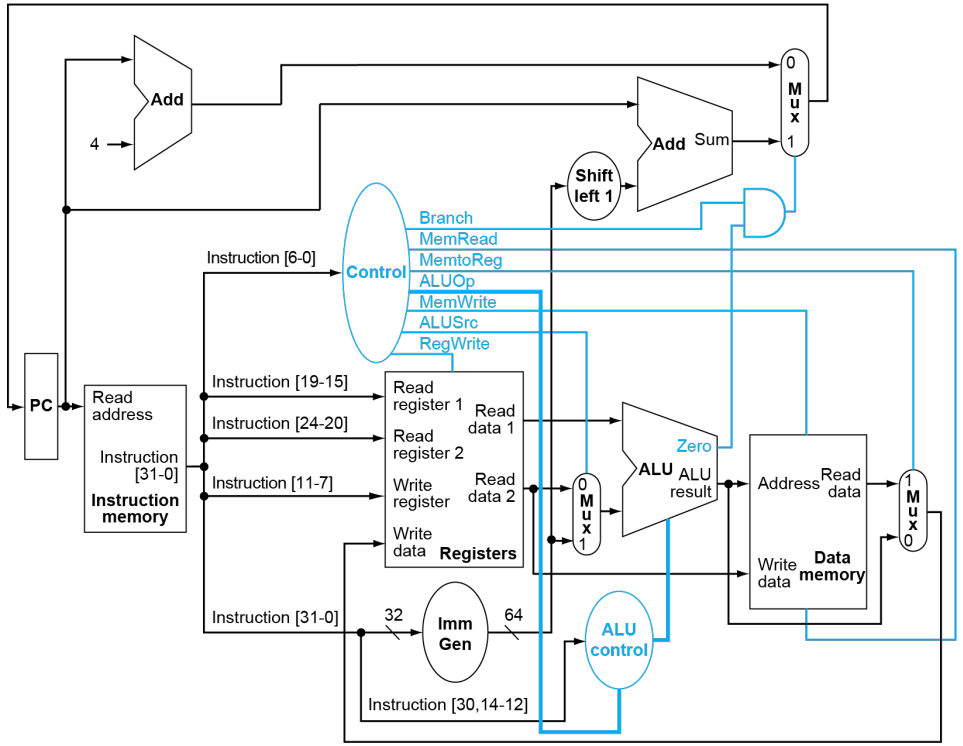
```
test ImmGen Success: 6 tests passed in 5 cycles taking 0.039271 seconds

[info] [0.014] RAN 0 CYCLES PASSED
[info] ImmGenTest:
[info] ImmGen
[info] - should
[info] ScalaTest
[info] Run completed in 1 second, 669 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```

Task 4: Control

- The last task is to implement the main control unit that generates the seven control signals.
  - ➢Implement the module so that it generates control signals correctly for the same set of instructions as Task 3.

```
class Control extends Module {
  val io = IO(new Bundle{
    val in = Input(UInt(7.W))
    val write_reg = Output(Bool())
    val aluSrcFromReg = Output(Bool())
    val memWrite = Output(Bool())
    val memToReg = Output(Bool())
    val aluOp = Output(UInt(2.W))
    val branch = Output(Bool())
  })
…
}
```



Task 4: Control

- Example Output

```
sbt/bin/sbt "testOnly ImmGenTest"
```

```
test Control Success: 28 tests passed in 5 cycles taking 0.022416 seconds

[info] [0.012] RAN 0 CYCLES PASSED
[info] ControlTest:
[info] ImmGen
[info] - should
[info] ScalaTest
[info] Run completed in 1 second, 460 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```