

CSCE 3301

Mohamed Shalan, PhD

Time Line

2

- ~~□ MS1 (17/10): Names and task assignments~~
- MS2 (28/10): The datapath block diagram, as well as an alpha version of the CPU RTL model. Also, basic test cases to verify the CPU core RTL model. The RTL model must support all of the RV32I required instructions. At this stage, the 2nd constraint may not be implemented. Also, ebreak and ecall may not be implemented.
- MS3 (4/11): Add support for compressed instructions and the 2nd constraint.
- MS4 (11/11): Add simplified support for interrupts, ecall and ebreak. Also, add limited support for CSR instructions. More details will be given on how to implement those.

Constraints

3

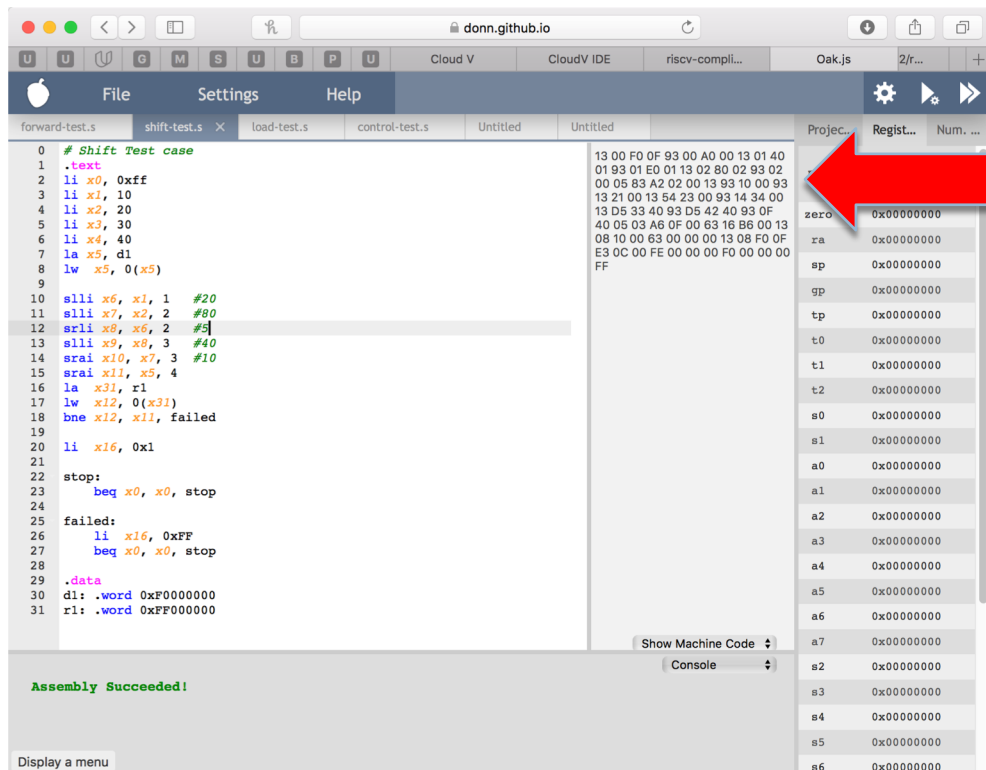
- 1) A single ported memory is used for both data and instructions.
- 2) A memory transaction is done in 2 phases.
- 3) A dual ported memory is used for the register file.

Tips

4

□ Developing “simple” testcases

1) Use oak.js (<https://donn.github.io/Oak.js/>)



The screenshot shows the Oak.js web IDE interface. The main editor displays assembly code for a test case. The code includes instructions like `li x0, 0xff`, `li x1, 10`, `li x2, 20`, `li x3, 30`, `li x4, 40`, `la x5, d1`, `lw x5, 0(x5)`, `slli x6, x1, 1 #20`, `slli x7, x2, 2 #80`, `slli x8, x6, 2 #4`, `slli x9, x8, 3 #40`, `srai x10, x7, 3 #10`, `srai x11, x5, 4`, `la x31, r1`, `lw x12, 0(x31)`, `bne x12, x11, failed`, `li x16, 0x1`, `stop: beq x0, x0, stop`, `failed: li x16, 0xFF`, `beq x0, x0, stop`, and `.data d1: .word 0xFF000000`, `r1: .word 0xFF000000`. The status bar at the bottom indicates "Assembly Succeeded!".

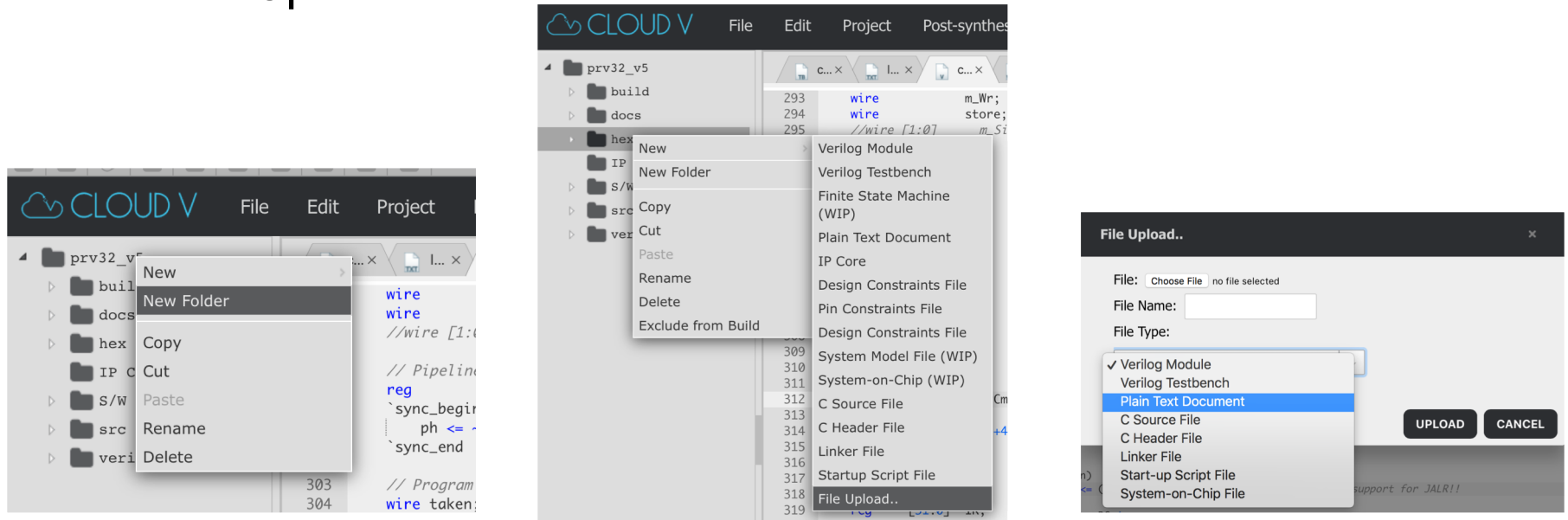
On the right side, there is a memory dump table. A red arrow points to the "zero" entry, which has a value of `0x00000000`. The table lists various registers and memory locations with their corresponding values.

Register/Label	Value
zero	0x00000000
ra	0x00000000
sp	0x00000000
gp	0x00000000
tp	0x00000000
t0	0x00000000
t1	0x00000000
t2	0x00000000
s0	0x00000000
s1	0x00000000
a0	0x00000000
a1	0x00000000
a2	0x00000000
a3	0x00000000
a4	0x00000000
a5	0x00000000
a6	0x00000000
a7	0x00000000
s2	0x00000000
s3	0x00000000
s4	0x00000000
s5	0x00000000
s6	0x00000000

Tips

5

- ❑ Developing “simple” testcases
 - 2) Save the machine code into a file (.hex extension)
 - 3) Upload into CloudV workspace
 - Create a folder named hex
 - Upload



Tips

6

□ Developing “simple” testcases

4) Load the hex file into your memory (Verilog array) using the system task \$readmemh()

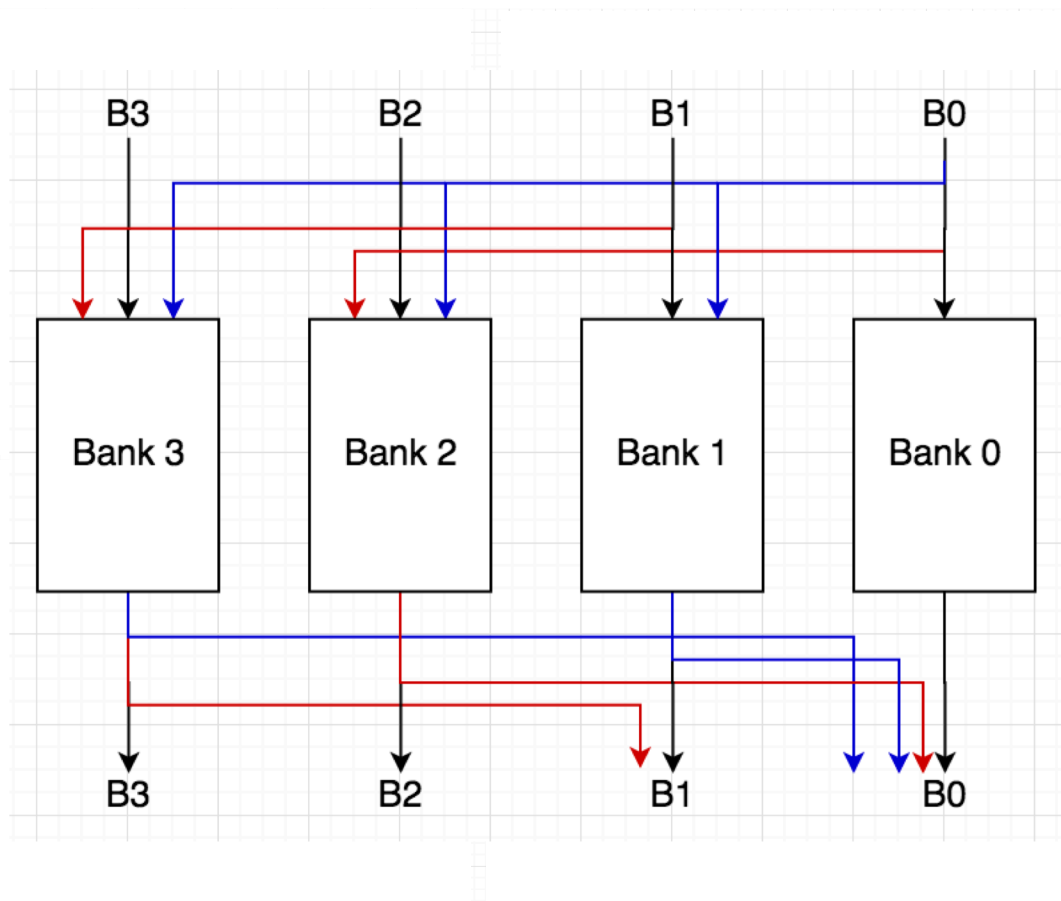
```
reg[7:0] mem[(4*1024-1):0]; // 4KB memory

initial begin
    $readmemh("./hex/test2.hex", mem);
end
```

Tips

7

- Memory support for different data sizes



Tips

8

□ Instruction Fetch and Compression

- Because of mixing 16-bit and 32-bit instruction words, you may end up with fetching a 32-bit word from a location which is divisible by 2 (not 4). Each half word comes from different words!
- If it is because the instruction that preceded this instruction is 16-bit and located @ a location which divisible by 4, you may always fetch one 32-bit instruction word every fetch cycle and save the unused half word to be used to construct the 32-bit word with the next fetch.
- If it is because of branching/jumping (transfer the control to a location which is not divisible by 4, you may fetch twice (stall the pipeline) to get the word. Other solutions are too expensive and degrade the performance!

Tips

9

□ Watch for this RAW Hazard

```
LW    x1, 0(x2)
```

```
SW    x1, 0(x4)
```

- The forwarding is from the pipeline register to the memory

How to implement GT, LE, GTU, LEU

10

- Subtract and look at the flags: Z, C, V and S
 - $EQ = Z$
 - $NE = \sim Z;$
 - $LT = (S \neq V)$
 - $GE = (S == V)$
 - $LTU = \sim C$
 - $GEU = C$