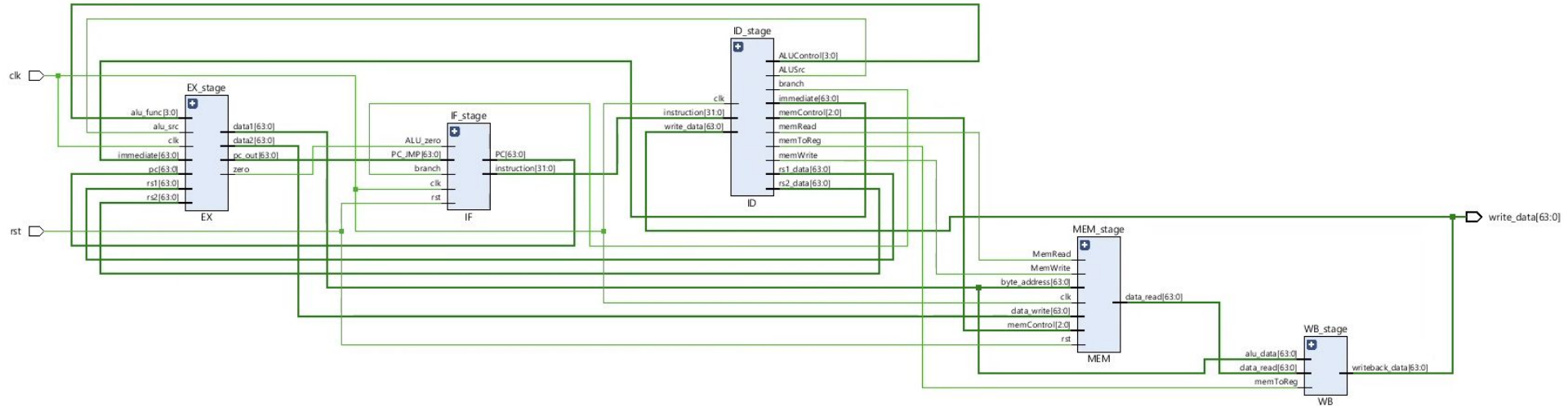
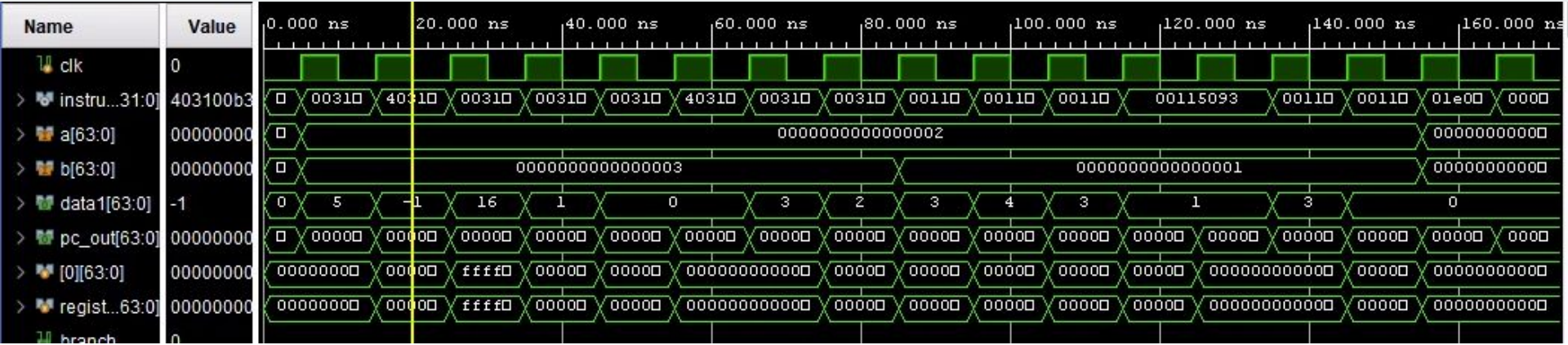
RISC V CPU

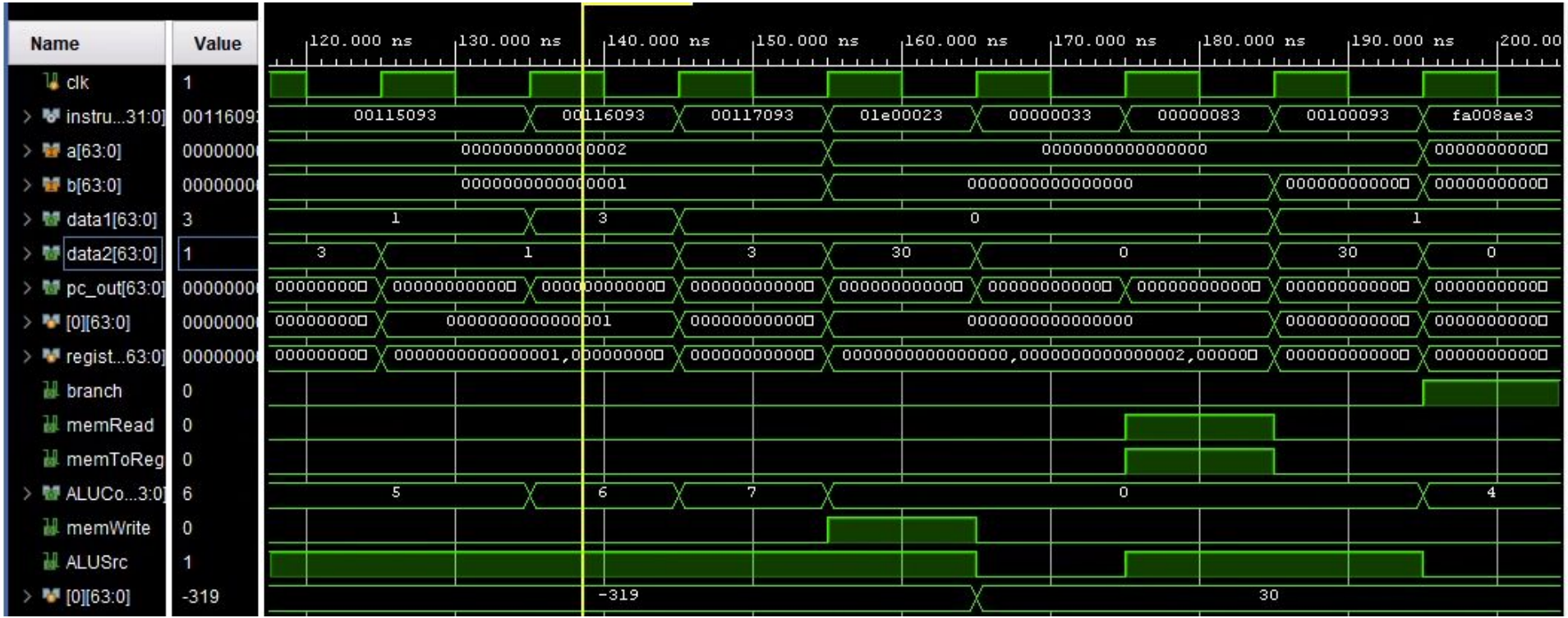
Jasper Edbrooke

Basic Single Cycle Stages

- IF
  - Fetch instructions from memory based on PC, handles basic branching logic
- ID
  - The Brain of the CPU, reads the instruction from IF and decodes it to send signals and data to rest of CPU
- EX
  - The Muscle of the CPU, does the main computation and execution of instructions
- MEM
  - Handles reading and writing to memory
- WB
  - Last stage to send data back into the registers from memory or previous calculations

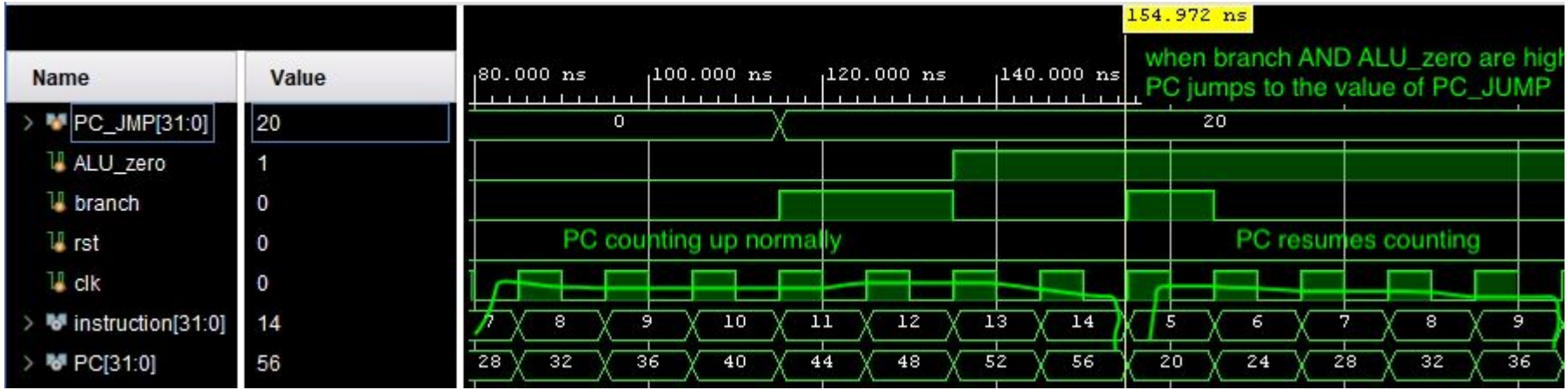

clk
rst
EX_stage
alu_func[3:0]
alu_src
clk
immediate[63:0]
pc[63:0]
rs1[63:0]
rs2[63:0]
data1[63:0]
data2[63:0]
pc_out[63:0]
zero
EX
IF_stage
ALU_zero
PC_JMP[63:0]
branch
clk
rst
PC[63:0]
instruction[31:0]
IF
ID_stage
clk
instruction[31:0]
write_data[63:0]
ALUControl[3:0]
ALUSrc
branch
immediate[63:0]
memControl[2:0]
memRead
memToReg
memWrite
rs1_data[63:0]
rs2_data[63:0]
ID
MEM_stage
MemRead
MemWrite
byte_address[63:0]
clk
data_write[63:0]
memControl[2:0]
rst
data_read[63:0]
MEM
WB_stage
alu_data[63:0]
data_read[63:0]
memToReg
writeback_data[63:0]
WB
write_data[63:0]


Name
Value
clk
0
instru...31:0]
403100b3
a[63:0]
00000000
b[63:0]
00000000
data1[63:0]
-1
pc_out[63:0]
00000000
[0][63:0]
00000000
regist...63:0]
00000000
branch
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
00115093
0000000000000002
0000000000000003
0000000000000001
ffff


Name
Value
clk
1
instru...31:0]
00116091
a[63:0]
b[63:0]
data1[63:0]
3
data2[63:0]
1
pc_out[63:0]
[0][63:0]
regist...63:0]
branch
0
memRead
0
memToReg
0
ALUCo...3:0]
6
memWrite
0
ALUSrc
1
[0][63:0]
-319
120.000 ns
130.000 ns
140.000 ns
150.000 ns
160.000 ns
170.000 ns
180.000 ns
190.000 ns
200.00
00115093
00116093
00117093
01e00023
00000033
00000083
00100093
fa008ae3
0000000000000002
0000000000000000
0000000000000001
0000000000000000
00000000000
1
3
0
1
3
1
3
30
0
30
0
000000000
0000000000000001
0000000000000000
0000000000000001,000000000
0000000000000000,0000000000000002,00000
-319
30

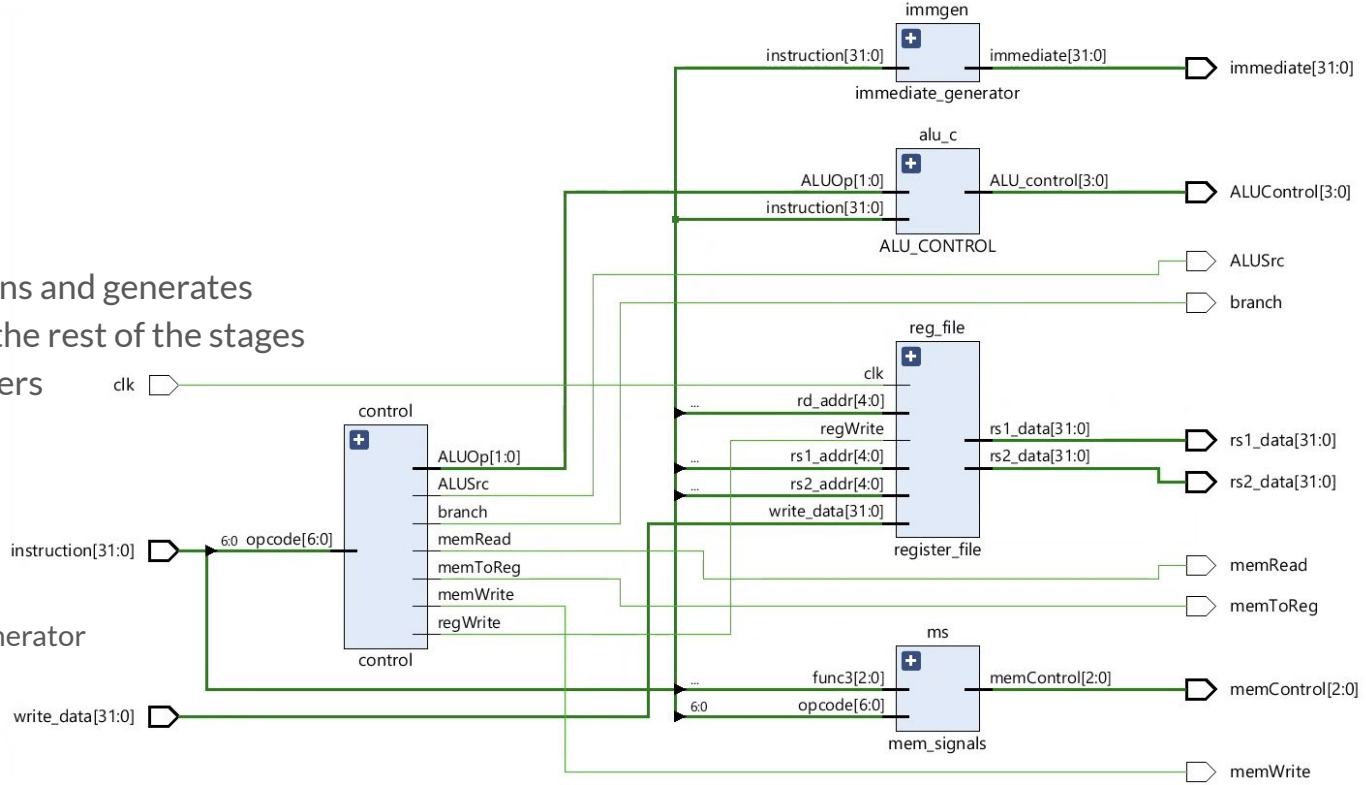
IF Stage

- Fetches instructions
- Handles branches



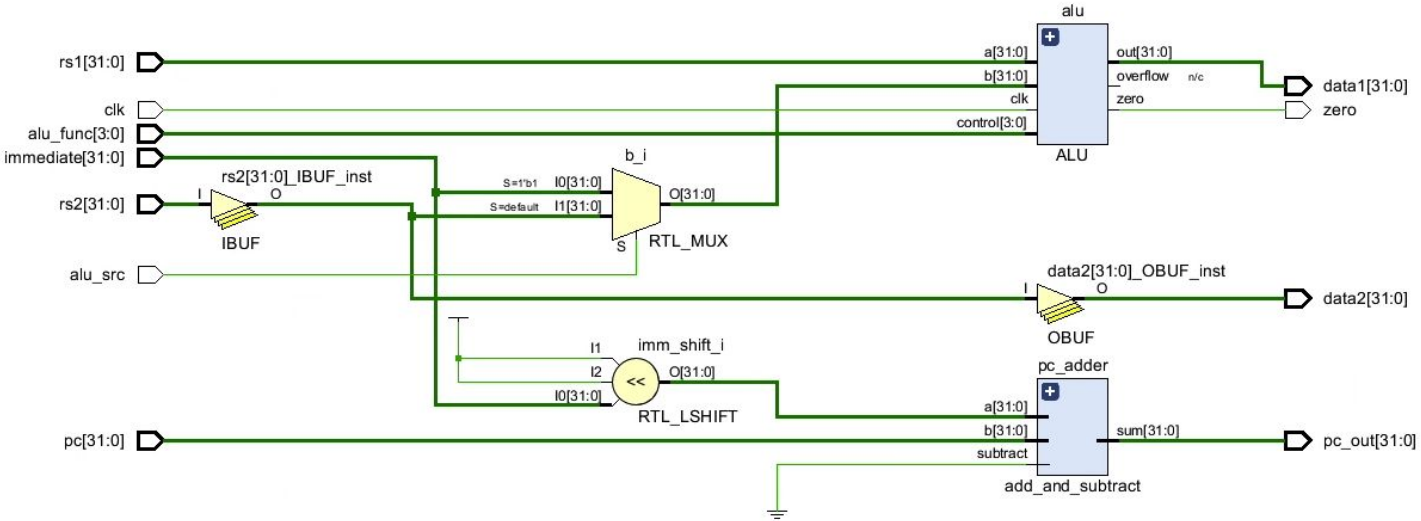
ID Stage

- Decodes instructions and generates control signals for the rest of the stages
- Handles the Registers
- Components
  - Register File
  - Control
  - ALU Control
  - Mem Control
  - Immediate Generator

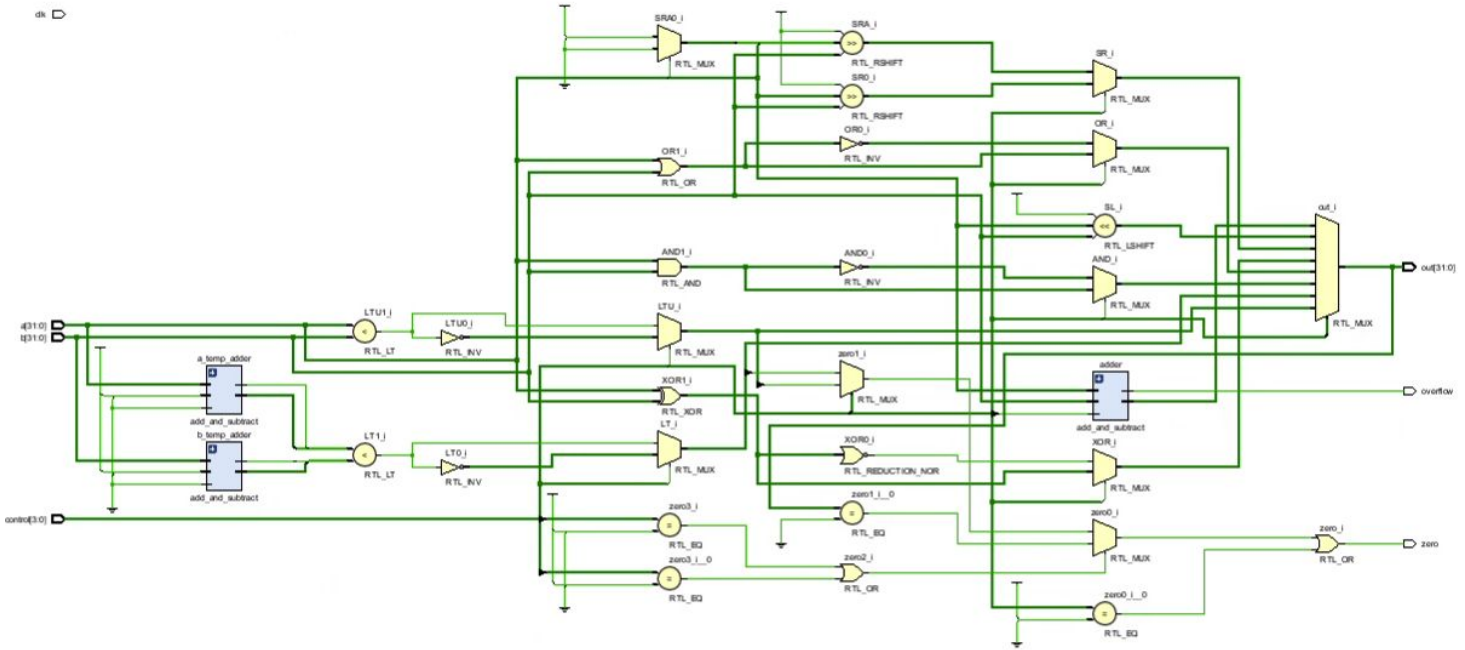


EX stage

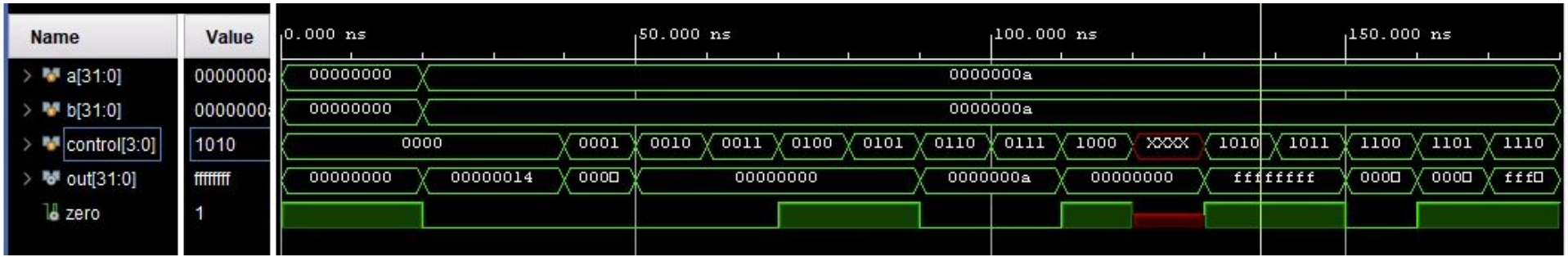
- Crunches the numbers in the CPU
- Takes signals from ID stage
- Components:
  - Main ALU
  - PC Branch adder
- Responsible for arithmetic instructions, and calculating address offsets for loads and stores



The ALU

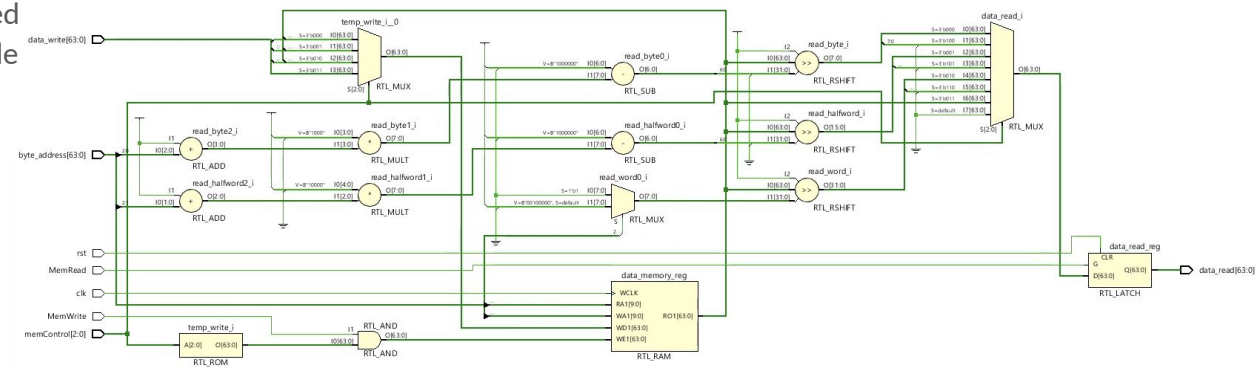


- Arithmetic and Logic unit
- Large combinational network
- Calculates all the possible operation, then control signal tells MUX which one the matches the instruction
- Outputs main Data, PC for branching, and Zero flag for branching


Name
Value
0.000 ns
50.000 ns
100.000 ns
150.000 ns
a[31:0]
00000000
0000000a
b[31:0]
00000000
0000000a
control[3:0]
1010
0000
0001
0010
0011
0100
0101
0110
0111
1000
XXXX
1010
1011
1100
1101
1110
out[31:0]
ffffffff
00000000
00000014
000□
00000000
0000000a
00000000
ffffffff
000□
000□
fff□
zero
1

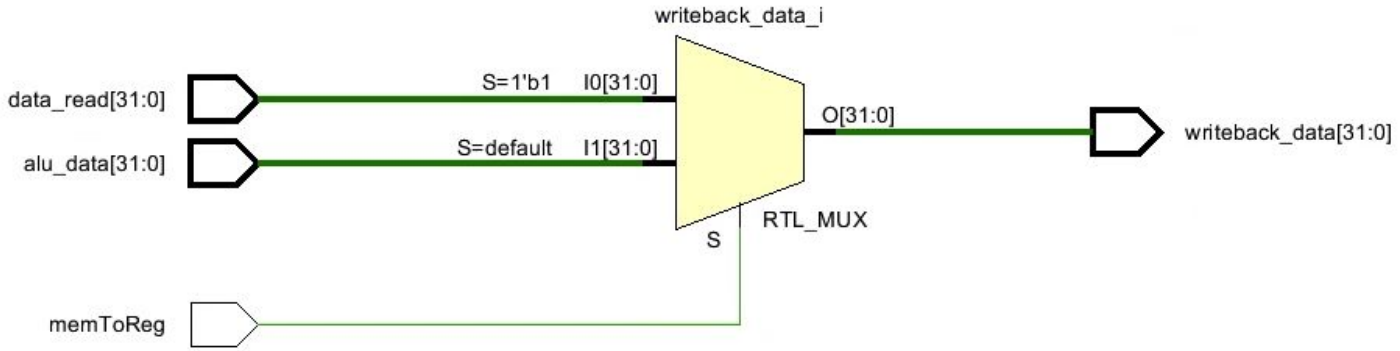
MEM Stage

- Handles reads and writes with Memory
- Limitations:
  - Writes must be 64-bit aligned to the beginning of a 64-bit dword
  - Reads are slightly less restricted, supports signed and unsigned reads:
    - Bytes can be from any byte
    - Halfword must be bits 0-15, 16-31, 32-47, or 48-63
    - Word must be 0-31 or 32-63
    - Dword must be 64 bit aligned
  - Each memory location is 64 bits wide

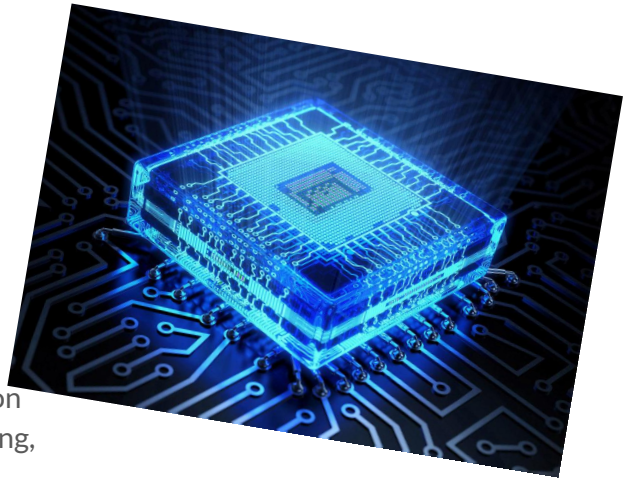


WB Stage

- Sends Data back to registers in ID Stage
- Basically just a mux between memory data and ALU data



Possible Improvements:



- Replace ripple carry adders in ALU with faster look ahead adders
- Implement proper pipelining, right now it is single cycle
  - But each stage is organized in such a way to be ready for pipelining
  - Would need the extra pipeline registers and hazard detection logic
- Add multiple issue slots, potentially an issue for loads and stores
  - Do a load/store operation in the same cycle as an R or I type instruction
  - I/R types don't access memory, and load/Stores only need alu for adding, so put in a separate issue slot with a simple adder

Thanks!
Time for some Q & A