RegWritreProject C

Prelab

IFetch:

- RST
- PC
- PCplus4
- instruction

Dec:

- Branch
- RegDst
- Jump
- JumpReg
- EqNe
- LtGt
- LSSigned
- ALUOp
- Rt
- Rs
- PCplus4

Exec:

- ALUSrc
- JumpLink
- ALUOp
- RegOut1
- RegOut2
- SignExtendedImm
- AluOut

Mem:

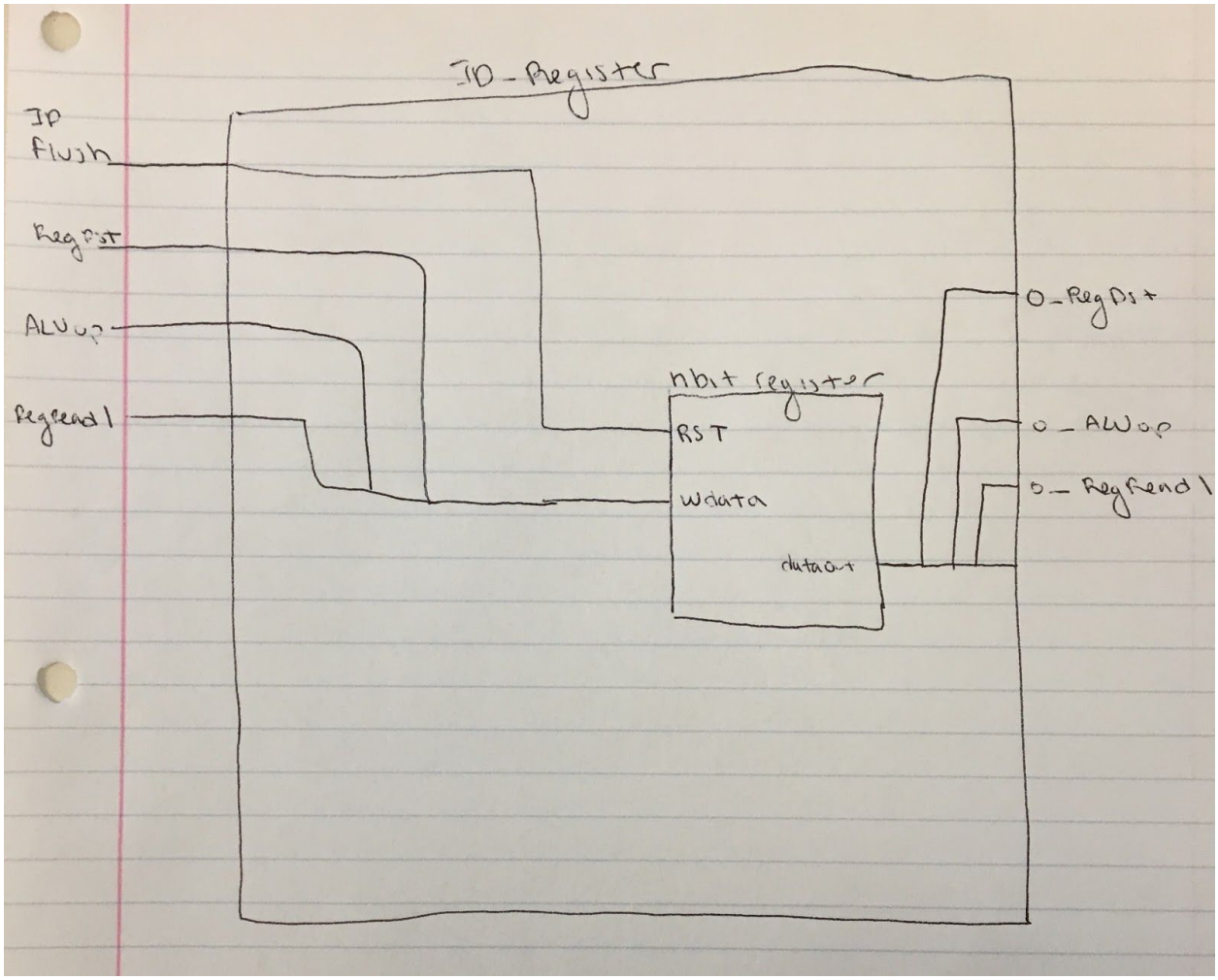
- MemWr
- MemSign
- MemHW
- MemByte
- ALUOut
- Rd

WB:

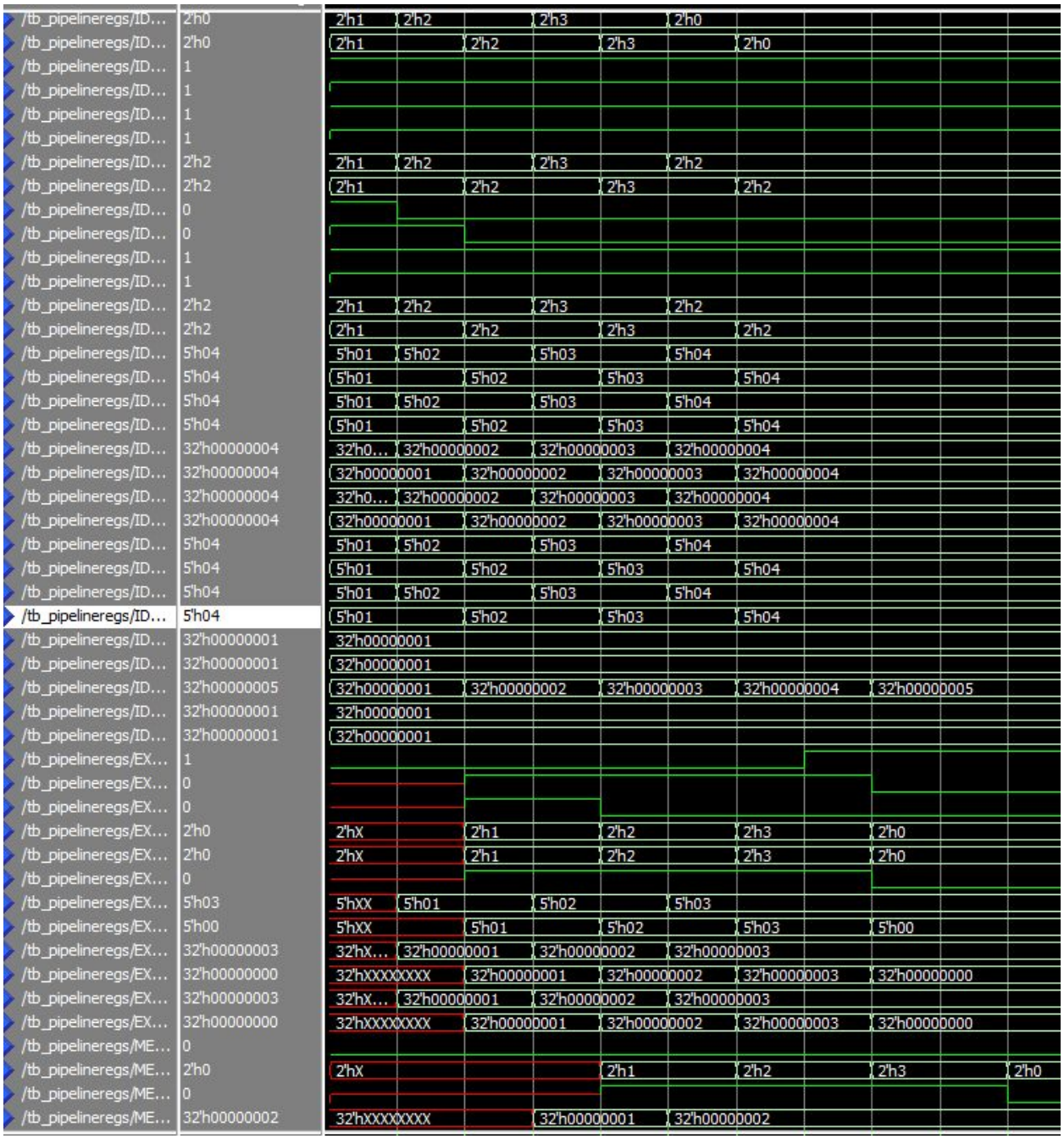
- DatatoReg
- RegWr
- MemOut
- Rd
- ALUOut

Part 1

a)


ID - Register
IF Flush
RegDst
ALUop
RegRead1
nbit register
RST
Wdata
dataout
O - RegDst
O - ALUop
O - RegRead1

b)



In this testbench, you can see values fill up one register, and then move to the next register at the clock cycle. The last register, MEM/WB, is the last register to get the values from the first instruction. In the last cycle, the reset functionality is demonstrated on only the EX/MEM register.

Part 2

a) Add, addu, addi, addiu, sub, subu, and, andi, or, ori, nor, xor, xori, mul, sll, srl, sra, sllv, srlv, srav, lui ← ALU operations

Jal, jalr ← comes from instruction counter

Lb, lbu, lh, lhu, lw ← come from memory

These instructions all write back to the registers. The signal RegWrite will be 1.

Sb, sh, sw

These are instructions that produce, or write to Dmem. They will cause MemWrite to be 1.

b) Add, addu, addi, addiu, sub, subu, and, andi, or, ori, nor, xor, xori, mul, sll, srl, sra, sllv, srlv, srav, Sb, sh, sw, Lb, lbu, lh, lhu, lw

These instructions consume some value. In these instructions, RegRead will be 1. In load instructions, MemRead will also be 1.

c) **Data dependencies (in general):** Branch, Jump, Load-Use, and read after write hazards

Data dependencies requiring forwarding: Any time an instruction that reads a value that is being written to in an instruction right before it, or two instructions before it. The exception is if the register being written to, and then read from is register zero.

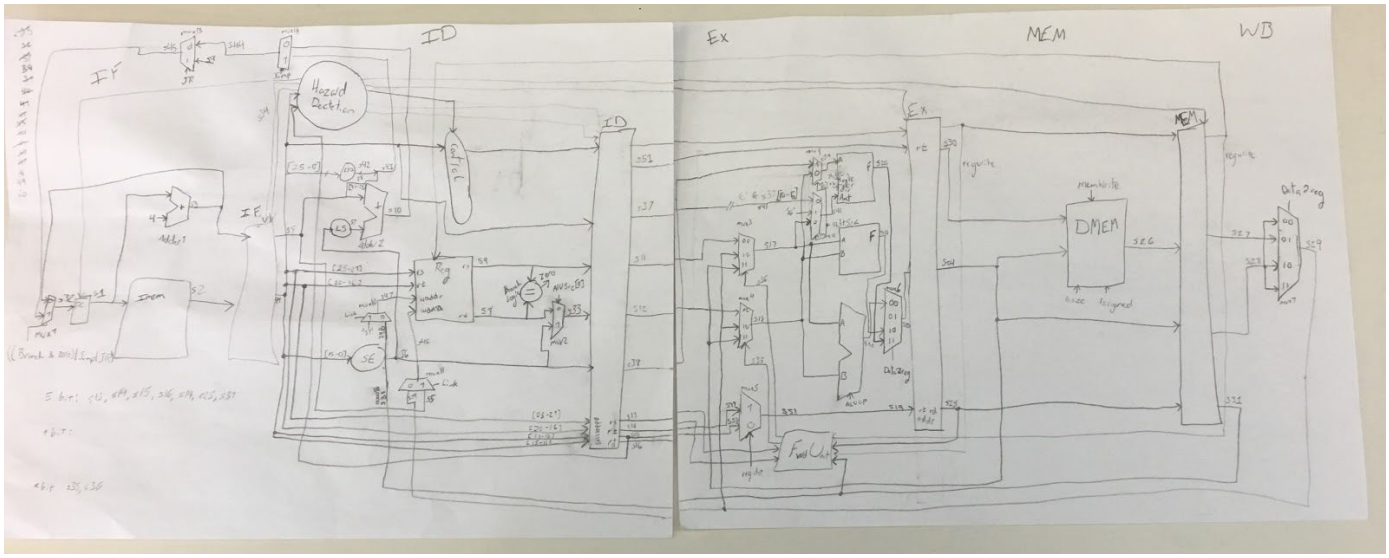
Data dependencies requiring hazard stalls: Branch, Jump, Load-Use

Part 3

a) Data forwarding
   - **i)** if(MEM_RegWrite = 1 && EX_RD != 0 && EX_Rd = ID_Rs) : ForwardA = 11
   - **ii)** if(MEM_RegWrite = 1 && EX_RD != 0 && EX_Rd = ID_Rt) : ForwardB = 11
   - **iii)** if(WB_RegWrite = 1 && MEM_RD != 0 && MEM_Rd = ID_Rs && ! (MEM_RegWrite = 1 && EX_RD != 0 && EX_Rd = ID_Rs)) : ForwardA = 10
   - **iv)** if(WB_RegWrite = 1 && MEM_RD != 0 && MEM_Rd = ID_Rt && ! (MEM_RegWrite = 1 && EX_RD != 0 && EX_Rd = ID_Rt)) : ForwardB = 10

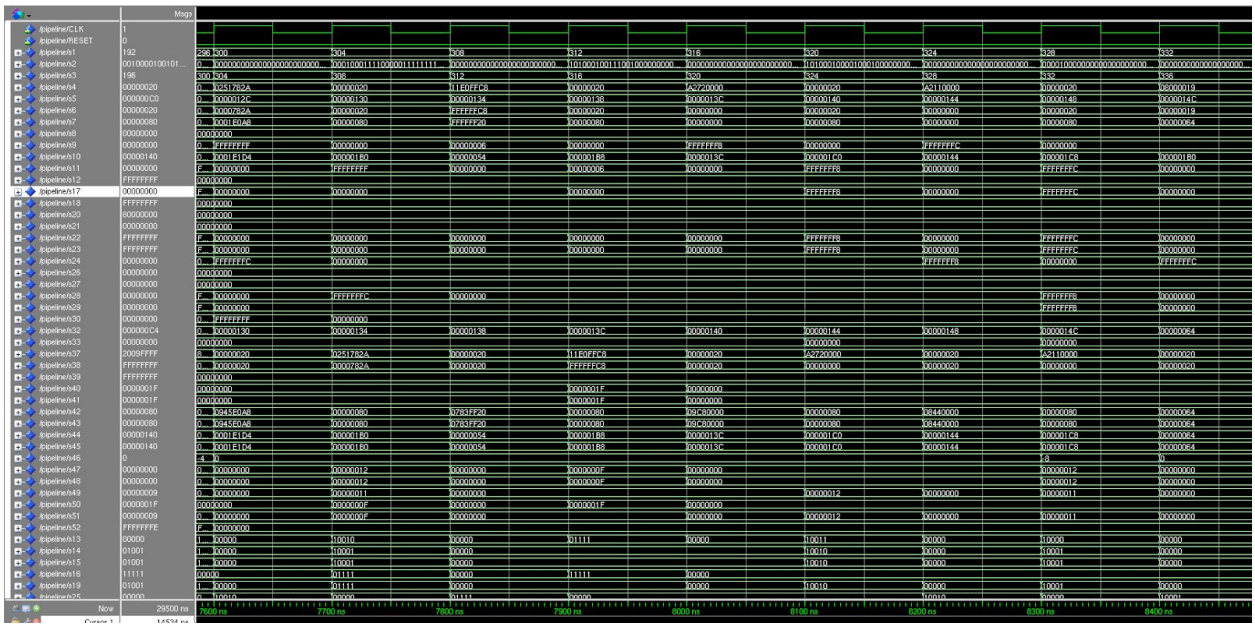
b) Hazard detection
   - **i)** if(ID_MemRead = 1 && (ID_Rt = IF_Rs || ID_Rt = IF_Rt)) : LoadUseHazard = 1
   - **ii)** if(Branch = 1 or Jump = 1) : BranchJump_Hazard = 1

Part 4



Part 5

a) We first tested our program that used all of our instructions with 4 noops between each original instruction. This was to make sure our pipeline registers and signals all worked without any hazards interfering. Next, we slowly removed the noops to ensure that the program would still work once hazards and forwarding came into play. We feel that our forwarding unit works as expected, but something about the hazard detection is still not correct.



b) We feel our first program that tested all instructions thoroughly tested our pipeline processor's forwarding and hazard detection units.

c) We tested our bubble sort next. It worked perfectly for our single cycle processor, but gave us some issues this time around. Our merge sort did not work in our single cycle processor, and unfortunately we were unable to get it into working condition for this pipeline processor.