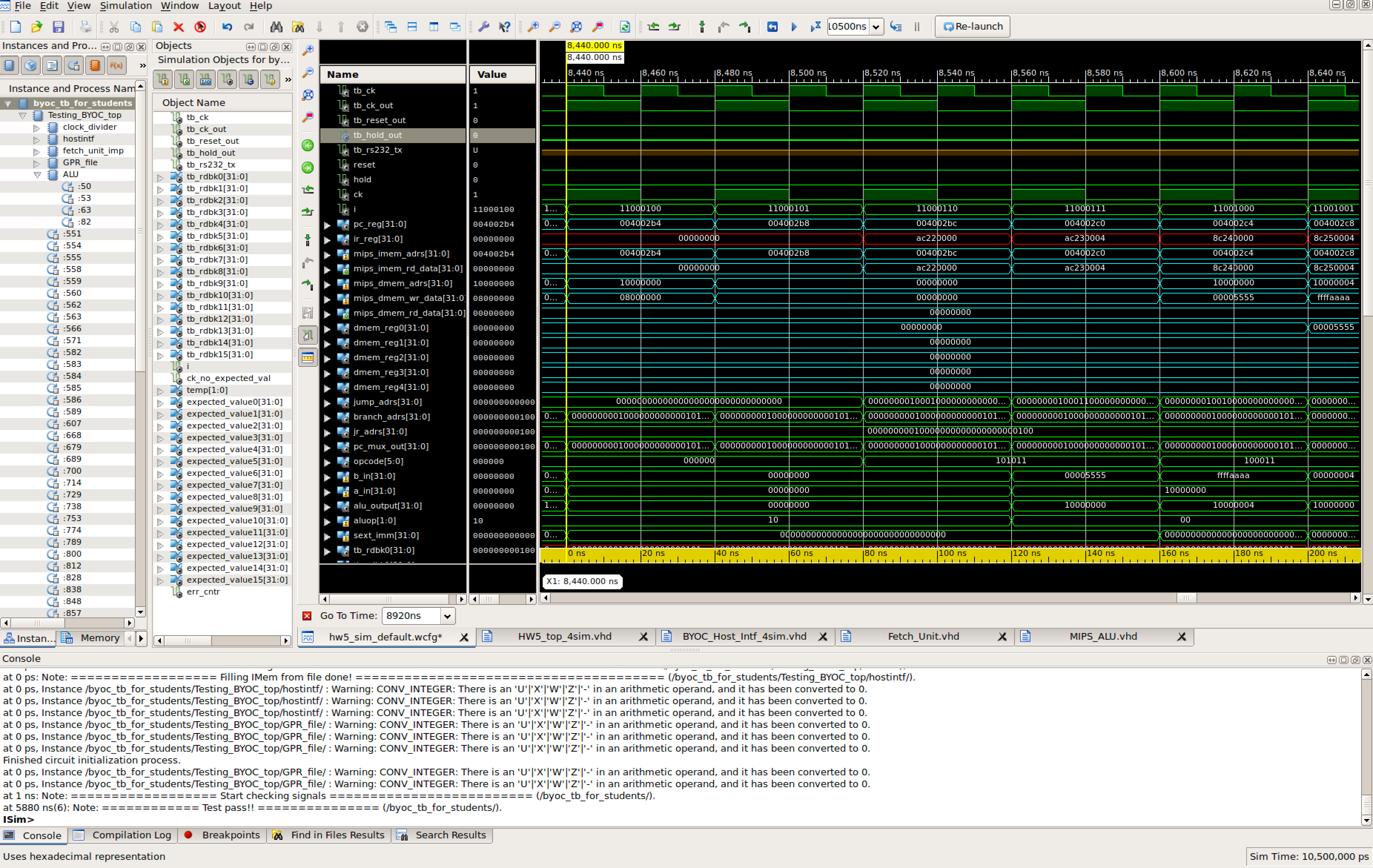
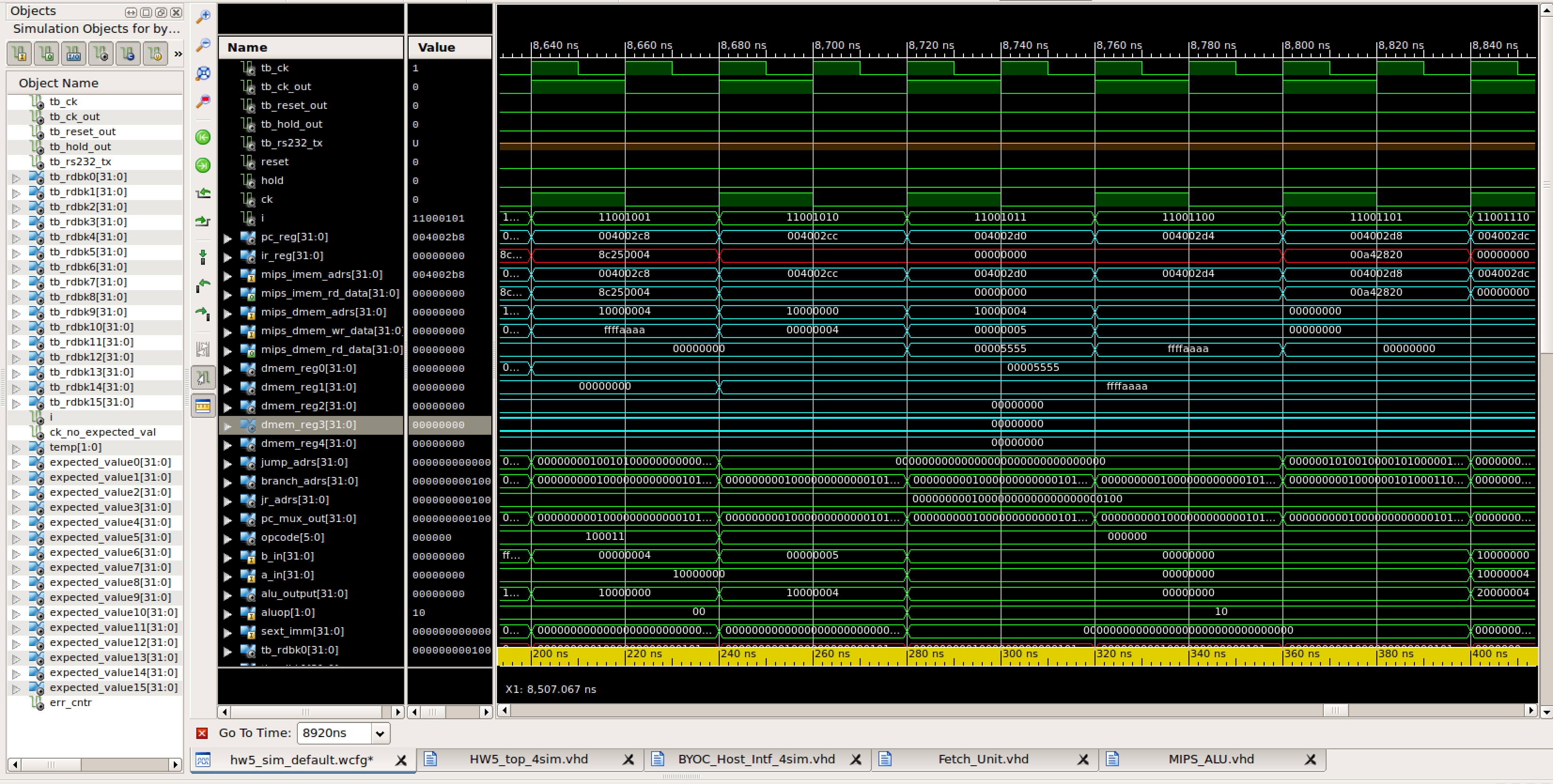
Homework #5 – Simulation Report

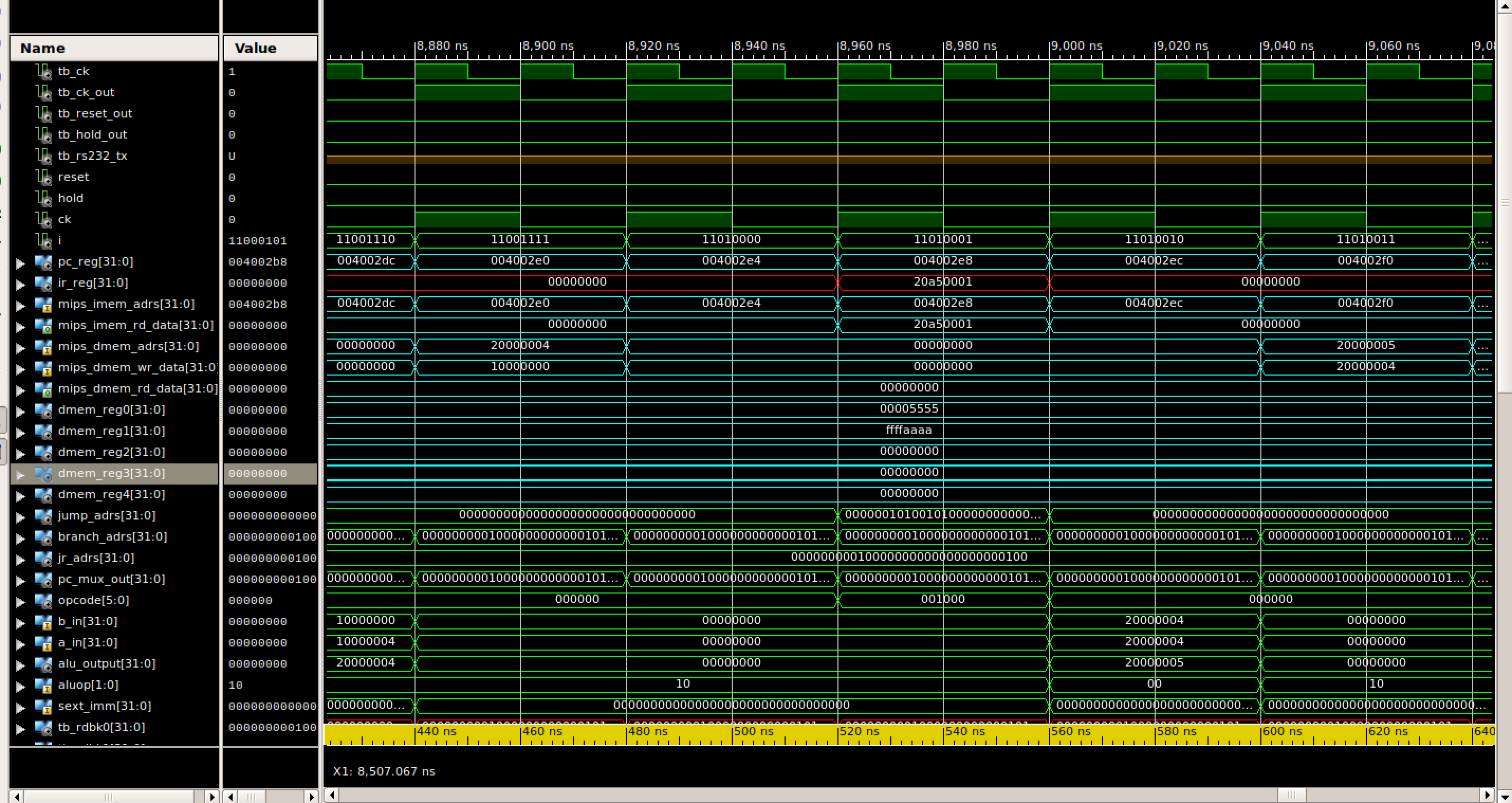
I.D. 204200026  
I.D. 201322708  
I.D. 300267390

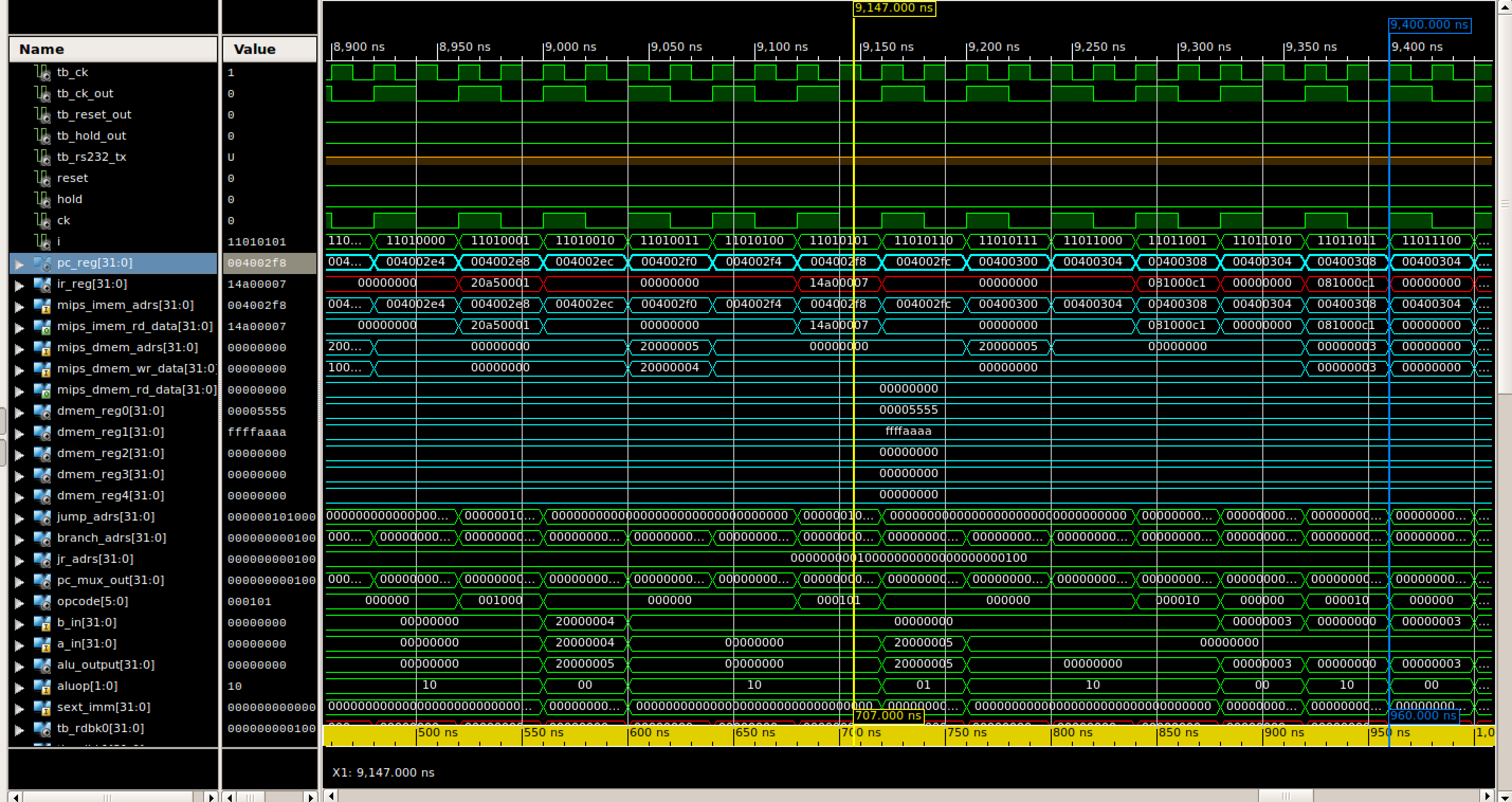
Group #1

3.2 Screen Captures of clock cycles 196 to 220









3.2. Explanation of signals (What is seen in the simulation)

As can be seen in the IR Register, the instructions issued between CK cycles 196 to 220 are as follows:

197 SW $2 $1 0

|  |  |  |  |
| --- | --- | --- | --- |
| 197 | 198 | 199 | 200 |
| IF | ID | EX | MEM |
| **SW $2 $1 0** | rt = $2, rs = $1 sext\_imm =0 | result of addition (in ALU output) -> 0x1000000 + 0 | DMEM\_adres : setting mem address 0x1000000 with the value  DMEM\_WR\_data : 0x5555 |

198 SW $3 $1 4

|  |  |  |  |
| --- | --- | --- | --- |
| 198 | 199 | 200 | 201 |
| IF | ID | EX | MEM |
| **SW $2 $1 0** | rt = $3, rs = $1 sext\_imm = 4 | result of addition (in ALU output) -> 0x1000000 + 4 | DMEM\_adrs : setting mem address 0x1000004 with the value  DMEM\_WR\_data : 0xffffaaaa |

199 LW $a4 $1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 199 | 200 | 201 | 202 | 203 |
| IF | ID | EX | MEM | WB |
| **Lw $4, $1 , 0** | Rt = $4 Rd = $1  Sext\_imm =0 | result of addition (in ALU output) -> 0x1000000 + 0 | Dmem\_adrs = 0x1000000 Dmem\_wr\_data – doesn’t write to it | Dmem\_wr\_data – doesn’t write to it  GPR\_wr\_data = 0x5555 Rd\_pWD = 4 ($4) |

200 LW $5 $1 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 200 | 201 | 202 | 203 | 204 |
| IF | ID | EX | MEM | WB |
| **Lw $5, $1 , 4** | Rt = $5 Rd = $1  Sext\_imm =4 | result of addition (in ALU output) -> 0x1000000 + 4 | Dmem\_adrs = 0x1000004 Dmem\_wr\_data – doesn’t write to it | Dmem\_wr\_data – doesn’t write to it  GPR\_wr\_data = 0xffffaaaa Rd\_pWD =5 ($5) |

201 nop

202 nop

203 nop

204 ADD $5 $5 $4 rd <- rs + rt

the same as previous exercise – this functionality has been tested

205 nop

206 nop

207 nop

208 ADDI $ $a1 0x0001

the same as previous exercise – this functionality has been tested

209 nop

210 nop

211 nop

212 BNE $a1 $zero 0x0007

213 nop

214 nop

215 nop

216 J 0x01000C1

217 nop

218 J 0x01000C1

…

Since we already tested previous commands (add, addi and BNE) in the previous exercise and their control singals have not been modified, we will test the validity of our mem related instructions SW and LW.

3.3

As explained in the previous exercise, R-type instructions require at least 3 clock cycles between them (3 nop insturcitons) if the latter instruction relies on the result of the previous instruction. This is due to the fact that data being written to the registers has to “flow” through the MIPS phases (and registers), an additional cycle in comparison to the previous exercise, since we’ve added the MEM phase.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX | MEM | WB (of 1st R-Type Inst.) |  |  |  |  |
| 1 | IF | ID | EX | MEM | WB |  |  |  |
| 2 |  | IF | ID | EX | MEM | WB |  |  |
| 3 |  |  | IF | ID | EX | MEM | WB |  |
|  |  |  |  | IF | ID (Reading of values written by previous instruction) | EX | MEM | WB |

As can be seen above, the data that has to be *read* is only *written* after 3 clock cycles, requiring we “wait” 3 nop commands.

3.4.

Translation of for loop code:

Addi $3 $0 0 // set I =0

addi $5 $0 10 // set r5 = 10 (upper limit for comparison)

nop (to allow $5 to be set correctly)

loop:

nop (2nd nop)

nop (3rd nop)

beq $3 $5 , continue:

nop

**{ loop code }**

addi $3 $3 1

j loop (first inst. Acts as nop)

continue:

A BEQ statement testing the register value written by a R-Type command must wait 3 nop commands (see above question). In addition, after the command is fetched, decoded and executed, the jump only occurs after 3 clock cycles (because of the same considerations descrbied above regarding R-Type commands). Additionaly, when a beq instruction is issued, we must compare the Rs register to the Rt register specified in the instruction. This requires decoding, fetching and comparing the 2 in the ALU during which we would not like to continue running until the result of the comparison is obtained. Therefore, we must wait until the signal stating whether to increment the PC by 4 or jump to an offset is received (Rs\_equal\_rT). The result is obtained after the ALU compares (subs) the 2 values stored in the registers, by the end of the ID phase. Therefore – we can see that we have to wait 1 clock cycle:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BEQ | IF | ID | **EX** | MEM | WB |  |  |  |  |
| Nop | 1 | IF | ID | EX | MEM | WB |  |  |  |
| If no branch |  |  |  | IF | ID | EX | MEM | WB |  |
| … |  |  |  |  |  |  |  |  |  |
| OR ! Jmp inst. address |  |  |  | IF | ID | EX | MEM | WB |  |
|  |  |  |  |  |  |  |  |  |  |

3.5.

Yes. The same limitations apply to the lw and addi instructions as they both utilize the writing phase(WB), therefore the same NOP instructions are required just like R-Type commands. As to the BNE, it has the same restrictions that a BEQ instruction has.

Possible solutions

1. S/W solutions:
   1. Using Nops for lw, addi, bne instructions (3 nops just like R-type commands since we are using WB phase and 1 nop for BNE)
   2. For branching instructions:

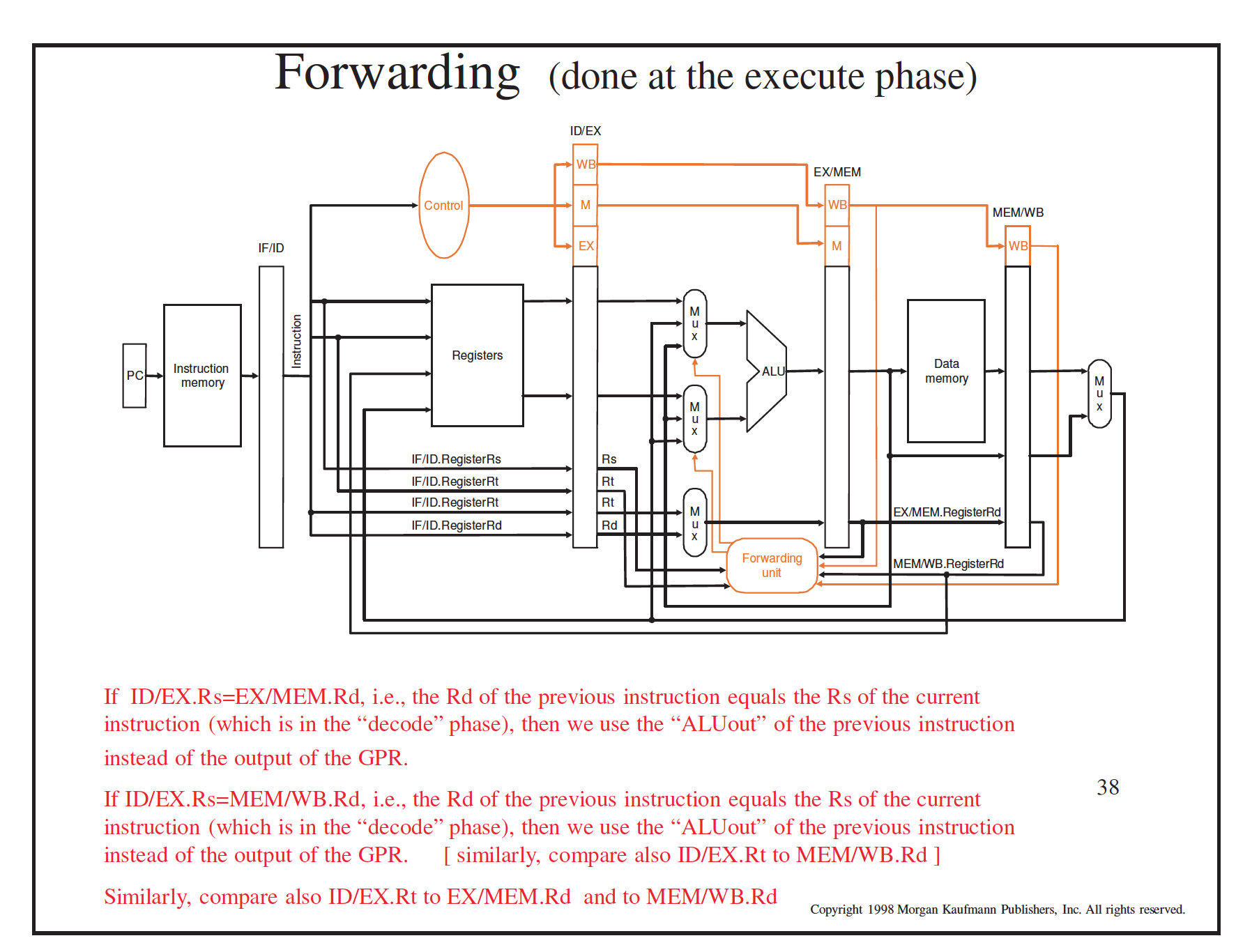
Predict - See the future commands and try to predict whether or not a branch will occur (for bne and beq intsructions)

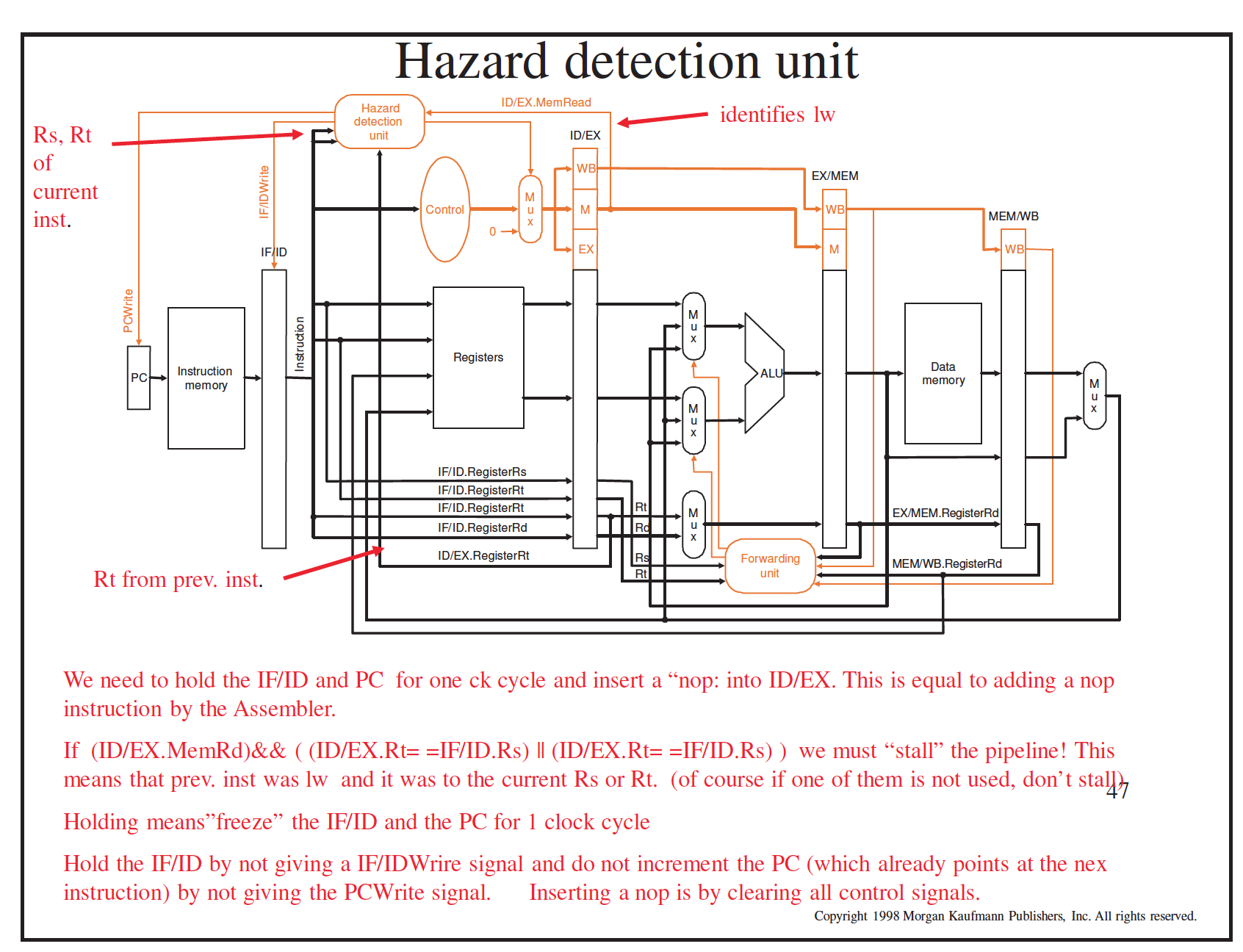
For example: always assume a branch will not occur and increment the PC normally. Only in cases of a branch we will fix the behavior and continue (at thend of the ID phase). If this happens we will need to discard the instruction that was already fetched and fetch the new instruction from the branch address into the IR (one cycle cost). This way we can gain one cycle of performance. (discarding can be done with resetting).

* 1. Insert real operations that can be performed instead of nop commands (without risking hazards – requires checking before/during compilation)

1. H/W solutions

Since this has been taught and mentioned in the digital architectures course, we retrieved soltuions presented by the literature of the previous course.

* 1. Forwarding:  
     however this does not solve the problem for the lw instruction. As stated in the course documentation
  2. Adding a Hazard detection unit  
     (also shown in the digital architectures course)



This will allow us to overcome the problem with **lw command**.

* 1. To solve the branching problem VIA hardware we might use “Flushing” to delete the instructions.  
     