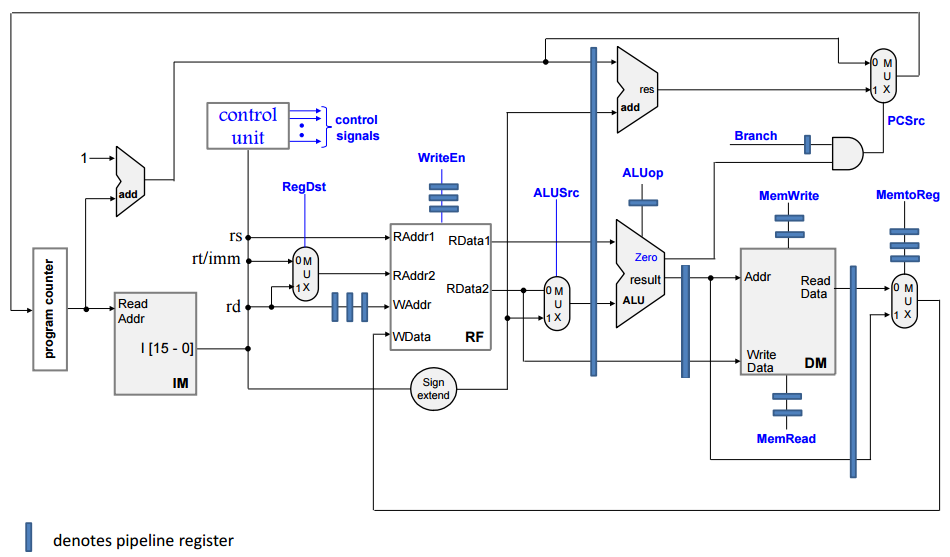
|  |
| --- |
| C:\Users\Gerald\AppData\Local\Microsoft\Windows\INetCache\Content.Word\NTULogo.png |
| Project Report |
| CZ3001 – [ADVANCED COMPUTER ARCHITECTURE](https://ntulearn.ntu.edu.sg/webapps/blackboard/execute/launcher?type=Course&id=_49817_1&url=) |
|  |
|  |
|  |
|  |

|  |  |
| --- | --- |
| Done by: | TAO PING  WANG LIPENG  LEE YURU SAMUEL  SING SWEE YANG |
| Lab group: | SSP1(Group 18) |
| Date submitted: | 11/11/2015 |

# Part 1

1. Five stage pipelined register:



Mux3

Adder1

Figure 1.0 5-stage pipelined processor

As shown in Figure 1.0, 4-stage pipelined processor is converted to the 5-stage pipelined processor which includes 3 pipeline registers in the datapath at the ID/EXE, EXE/MEM and MEM/WB interface.

For the five stage pipelined processor:

* During the first cycle: fetch instruction
* During the second cycle: decode instruction
* During the third cycle: execute instruction with the data retrieved in the ALU module
* During the fourth cycle: a memory read or write will be done if required in the data memory
* During the fifth cycle: write back to the register file if write enable signal is passed into the register file.

Our task for part 1 is to implement three new instructions in this five-stage pipelined processor. The followings are the brief explanations for the three instructions:

* LW (Load Word) instruction

Function: fetches a word from memory, and loads into a register

Example: lw $rt, offset($rs) ($rt mem←[$rs + offset]

* SW (Store Word) instruction

Function: stores the content of a register in memory

Example: sw $rt, offset($rs) (mem[$rs + offset] ← $rt )

* BEQ (Branch On Equal) instruction

Function: branches if the content of two specified registers are equal

=Example: beq $rs, $rt, level

In order to implement these instructions into our design, there are additional modules need to be added.

* An adder (Adder1) to add immediate data and program counter.
* A multiplexer (Mux3) to select either nPC or the output of the adder (res)
* A branch implementation to decide when branch signal and output from ALU are equal
* Multiplexer (RegDst) to select raddr2 based on rd and imm value. This is for SW and BEQ instructions.
* Multiplexer (ALUsrc) to select either imm or rdata2 to ALU. This is for LW and SW instructions.
* Multiplexer (MemtoReg) to select either the output (result) from ALU or the output (ReadData) from data memory (DM). This is for LW instruction.

1. Test bench analysis

The test program consists of the instructions that specified in Figure 1.1. The initial register value defined is shown in Figure 1.2.

Based on the instructions and the initialised value, the following operations are performed:

* The content at memory location ([R2]+2) will be loaded at R6;
* The content at memory location ([R4]+1) will be loaded at R7;
* The content in R6 will be added to R7 and the result will be saved to R8;
* The content in R2 will be increased by 1
* The content in R5 will be increased by 1;
* The content in R8 will be stored at memory location ([R4]+4);
* The content in R4 will be increased by 1;
* If the content in R5 is equal to the content in R1, branch out of the loop

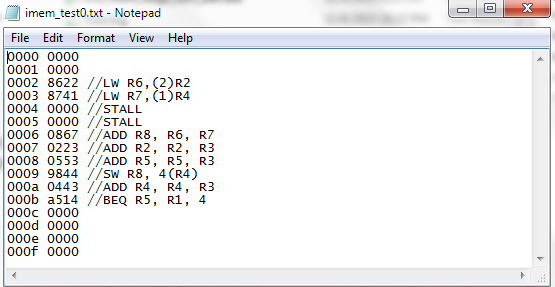


Figure 1.1 Instructions in test program

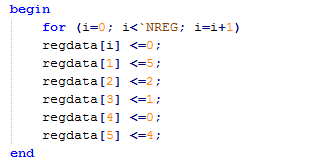


Figure 1.2 Register value defined

With the data memory shown in Figure 1.3, the test bench result is generated as shown in Figure 1.4. For LW instruction, data 0005 is loaded into register R6 from address 0004 (address 0002 offset by 2). Data 0002 is loaded into register R7 from address 0001 (address 0000 offset by 1). For SW instruction, data 0007 (0005 +0002) in R8 is stored to the address 0004 (address 0000 offset by 4). For BEQ instruction, when the content in R5 is equal to R1, jump by 4, which is 0010.

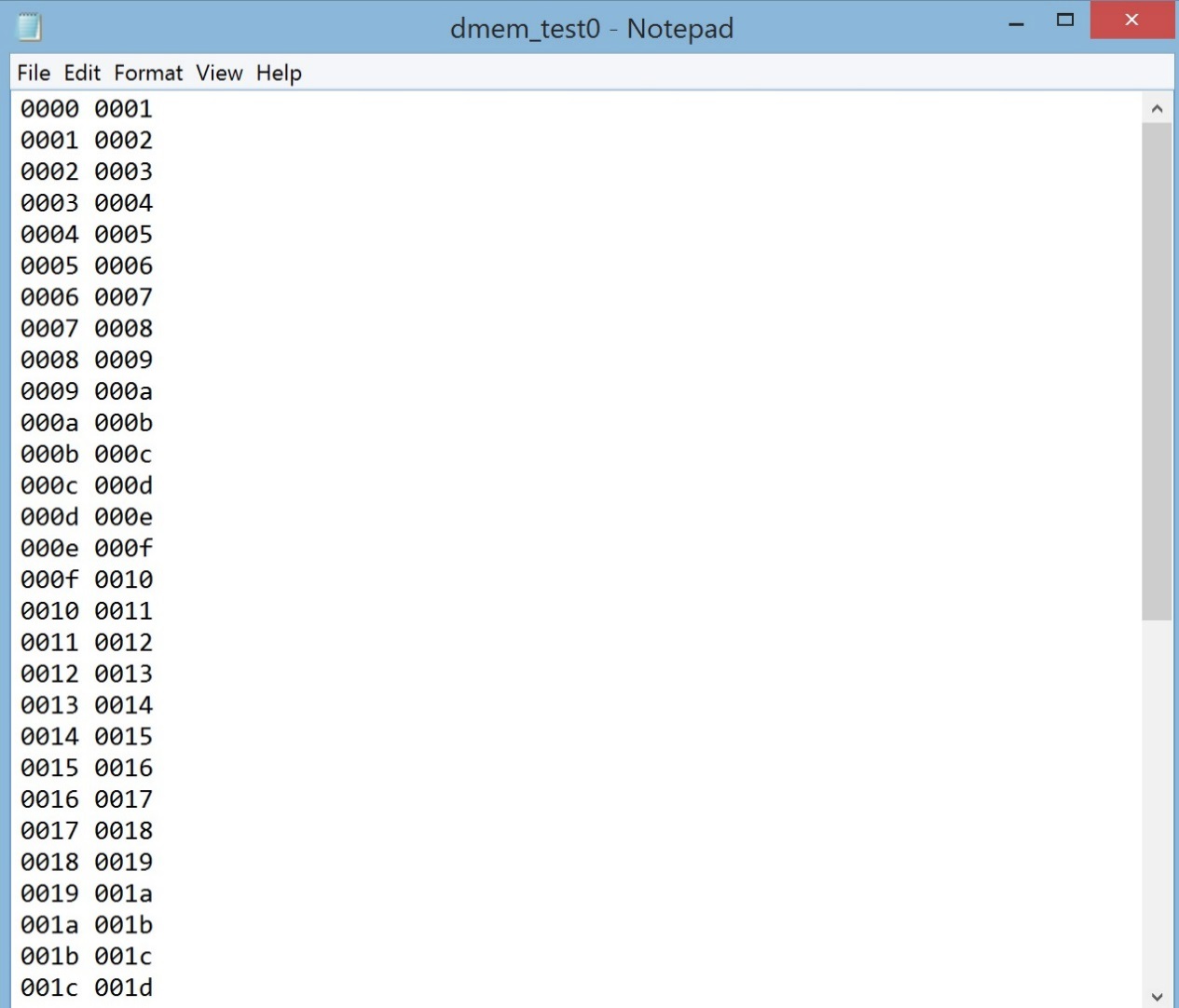


Figure 1.3 Data memory

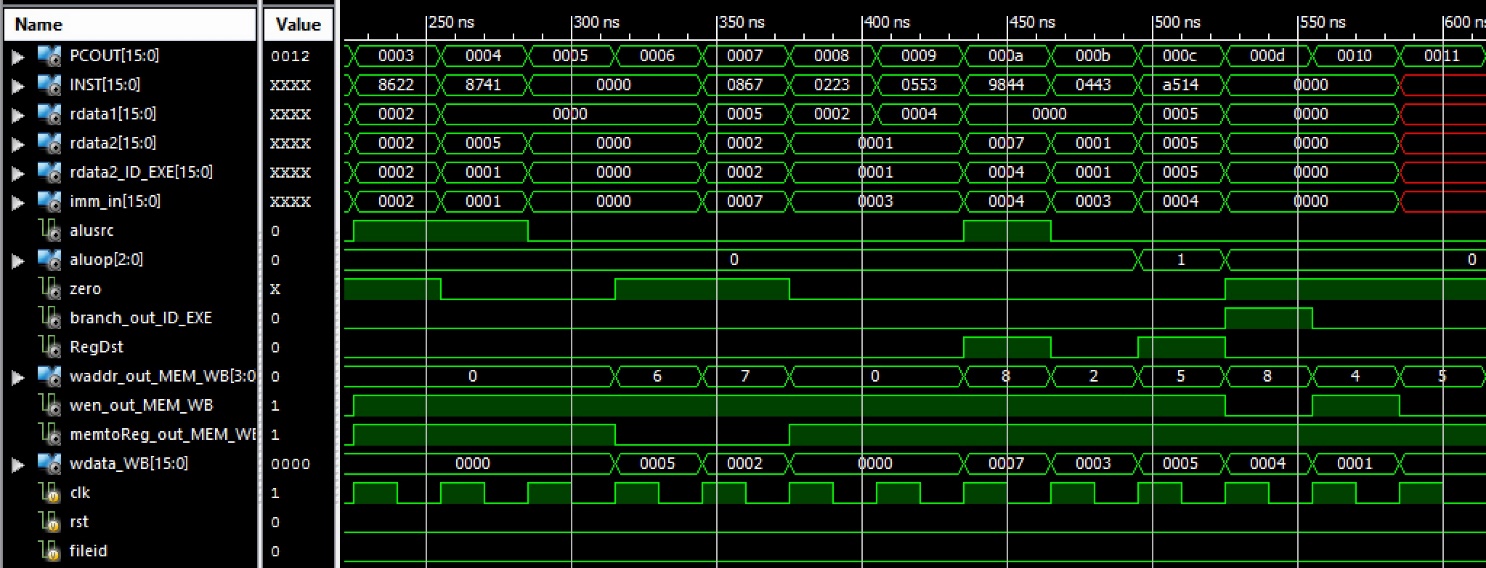


Figure 1.4 Test bench result

The Synthesis report result is shown in the following table:

|  |  |  |
| --- | --- | --- |
| Number of LUTs | Number of slice registers | Minimum period |
| 831 | 420 | 8.368 ns |

# Part 2­­

1. Test bench screenshots.

You should present the testbench screen shots for the execution of any program involves

J, JR and JAL instructions along with R & I instructions and part 1 instructions.

2. Explain the working of J, JR, JAL instructions in the same program.

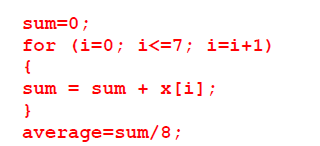
3. Synthesis report:

|  |  |  |
| --- | --- | --- |
| Number of LUTs | Number of slice registers | Minimum period |
|  |  |  |

# Part 3

## Section-1

1. Assembly code of the original program segment given in the assignment.



Assembly Code of Assignment 3(a):

//[R0] has been hardcoded to be 0

// It assumed that the save address and load address has been saved into the registers

ADD R1,R0,R0 // SET [R1] TO 0, [R1] = sum

ADD R2,R0,R0 //SET [R2] TO 0, [R2] = of i

ADD R3,R0,8 //max i value

LOOP:

LW R5,MEM[R4+0] //[R5] = x[i], assuming address of x[i] is in R4

ADDI R4,R4, 1 //Shift to next memory entry. x[i+1]

ADDI R2,R2,1 //Increase i by 1.

ADD R1, R1, R5 //sum = sum + x[i]

BEQ R2, R3, AVE //if (i == 8), exit loop

NOP

NOP

NOP

J LOOP //Otherwise, loop it until i reaches 8.

NOP

NOP

NOP

AVE:

SRL R1, 3 //Divide by using shift right logical 3 times (2^ 3 = 8)  
 NOP

NOP

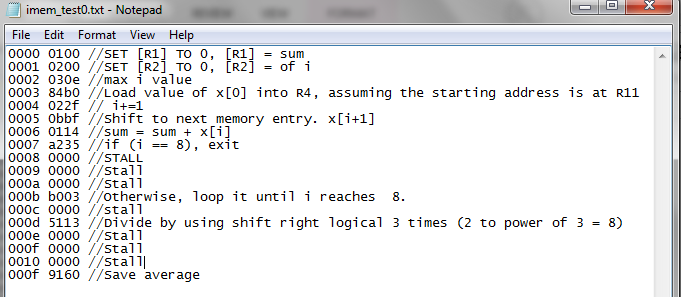
NOP

SW R1,MEM[R0+20] //Save average

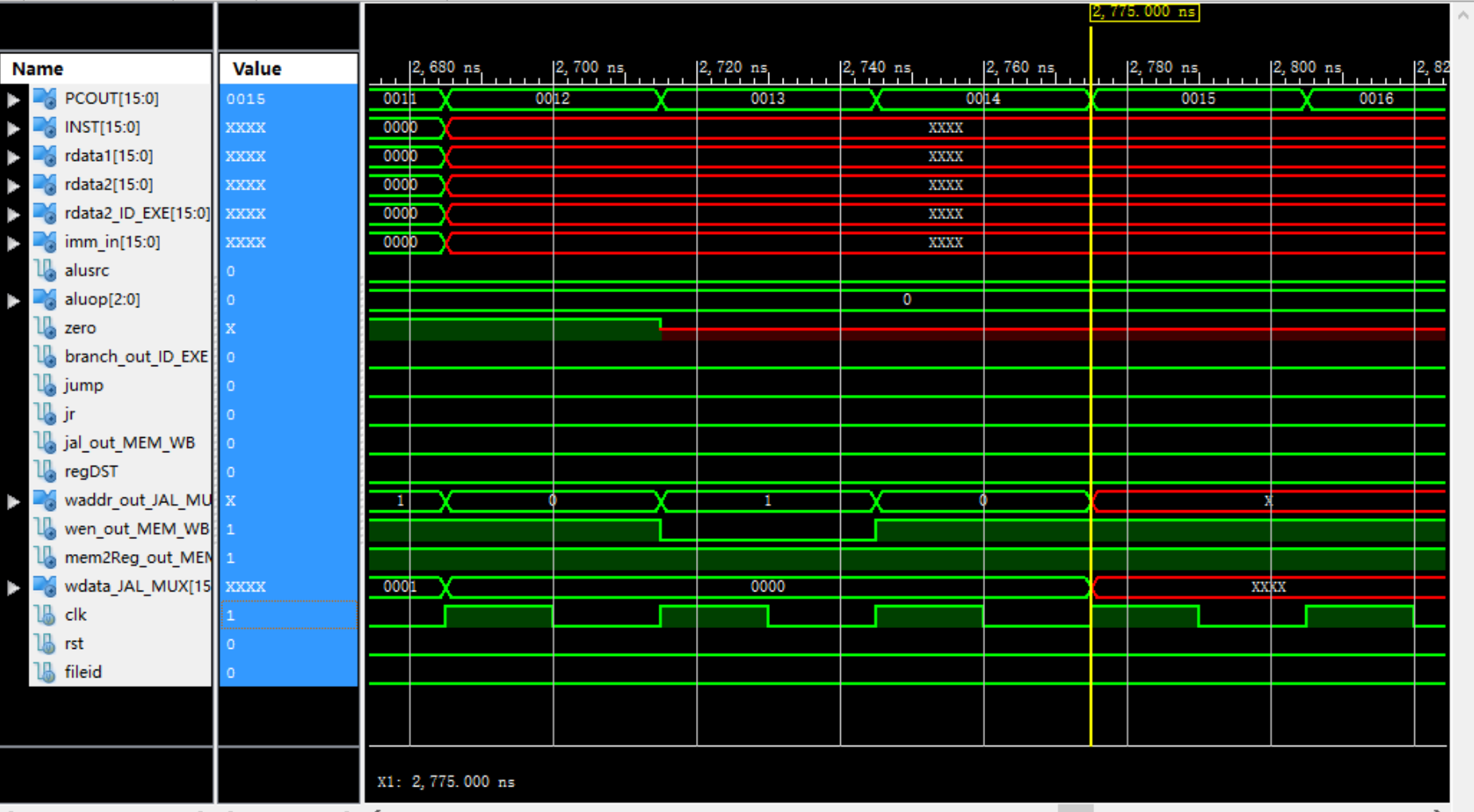
The assembly code is ran **4 times** to find average of a,b,c,d.

1. Machine Code

\*Reg[15] = 8, Reg[16]=1



3. Report the number of clock cycles taken for the execution of the program (where hazards are taken care of).



Number of clock cycles = (2775 – 135)/30 \*4  
 = 352

4. Execution time of the given program segment

Execution time for one array= ((2775)-135)

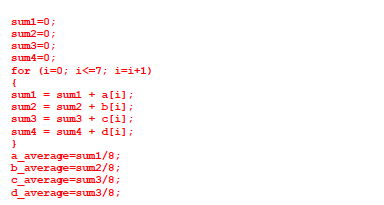
=2560

Execution time of program = 2560\*4

= 10560ns

## Section- II

5. Modification of the given program segment as mentioned in Part 3 of the question.



a. Modified assembly code of the given program segment.

//[R0] has been hardcoded to be 0

// It assumed that the save address and load address has been saved into the registers

ADD R1,R0,R0 //[R1]=sum1

ADD R2,R0,R0 //[R2]=sum2

ADD R3,R0,R0 //[R3]=sum3

ADD R4,R0,R0 //[R4]=sum4

ADD R5,R0,R0 // [R5] =current i value

ADDI R6,R0,8 // [R6]= 8 = max i value

LOOP:

LW R7, MEM[R10+0] //a[i] // R10 contains starting address of a

LW R8, MEM[R11+0] //b[i] // R11 contains starting address of b

LW R9, MEM[R12+0] //c[i] // R12 contains starting address of c

LW R10, MEM[R13+0] //d[i] // R13 contains starting address of d

ADD R1,R7,R1 // sum = sum1 + a[i]

ADD R2,R8,R2 // sum = sum2 + b[i]

ADD R3,R9,R3 // sum = sum3 + c[i]

ADD R4,R10,R4 // sum = sum4 + d[i]

ADDI R1,R1,1 //Shift to next memory entry. a[i+1]

ADDI R2,R2,1 //Shift to next memory entry. b[i+1]

ADDI R3,R3,1 //Shift to next memory entry. c[i+1]

ADDI R4,R4,1 //Shift to next memory entry. d[i+1]

ADDI R5,R5,1 //Increase i by 1.

BEQ R5, R6, AVE // Check if i ==8,Compute average if i reaches 8.  
 NOP

NOP

NOP

J LOOP //Otherwise, loop it until i reaches 8.  
 NOP

NOP

NOP

AVE:

SRL , R1, 3 //sum1 divide by srl 3 times (2 ^ 3 = 8)

SRL, R2, 3 //sum2 divide by srl 3 times (2 ^ 3 = 8)

SRL, R3, 3 //sum3 divide by srl 3 times (2 ^ 3 = 8)

SRL, R4, 3 //sum4 divide divide by srl 3 times (2 ^ 3 = 8) SW R1,MEM[R0+10] // save a\_average

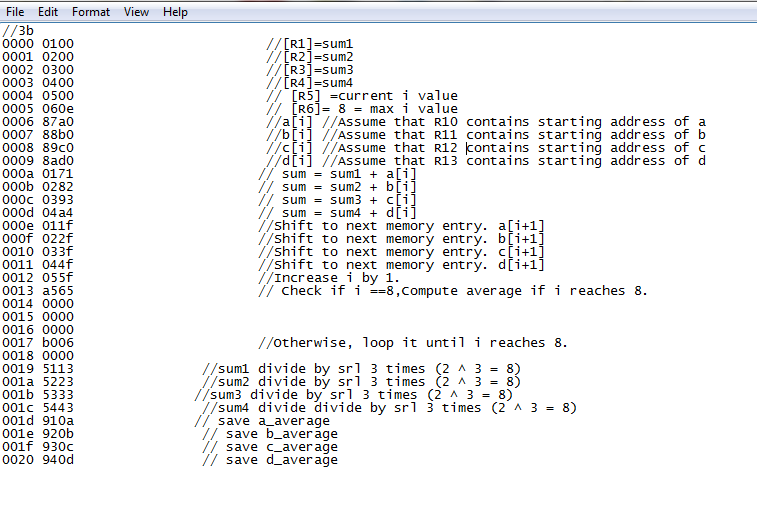
SW R2,MEM[R0+11] // save b\_average

SW R3,MEM[R0+12] // save c\_average  
 SW R4,MEM[R0+13] // save d\_average

The instructions have to been reordered to avoid data dependency hazard. For example, instead of using stalls, independent instructions are executed instead. As shown above, **after SRL R1,3**, 3 independent instruction are executed before **SW R1,MEM[R0+10]** instead of stalling. This avoids the data dependency (Read after Write) hazard without incurring extra clock cycles for stalls.

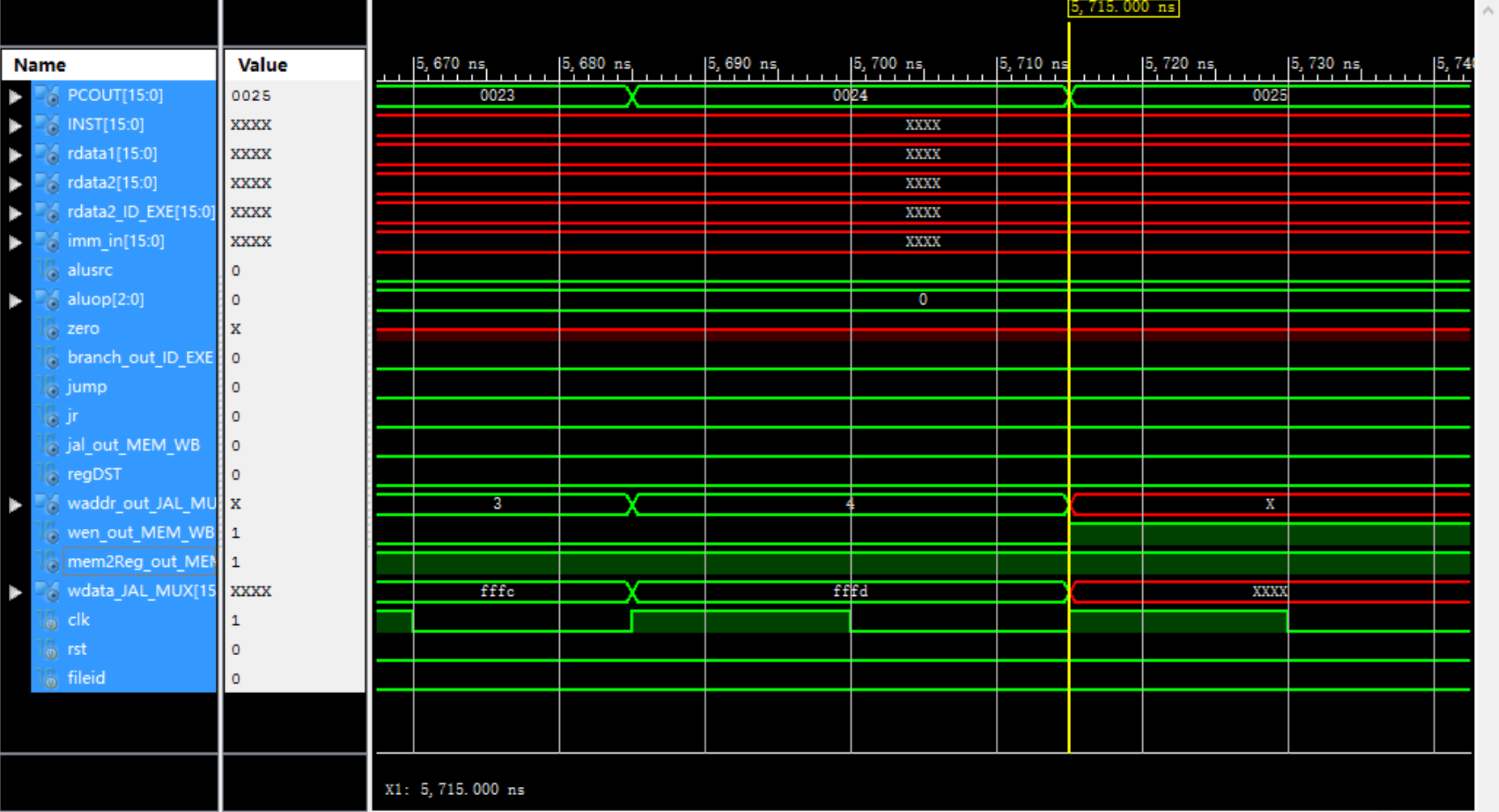
Also, the branch delay is reduced compared to the original code as there is total of the branch condition is checked 8 times with 1 branch delay (terminating condition, i == 8, once for all four arrays together) compared to the original code,where there the branch condition is checked 8\*4 times with 4 branch delays (terminating condition i==8, once for each array).

b. Machine code (The input to the Imem txt file).

\*Reg[15] = 8, Reg[16]=1

c. Report the number of clock cycles taken for the modified program segment (where

hazards are taken care of).



Number of clock cycles = (5715-135)/30

= 186

d. Execution time of the modified code.

Execution time = 5715-135

= 5580 ns

6. Speed up of modified code vs original code in Section 1

Speed up = 10560/5580

= 1.892

The significant speed up is due to the reduction of NOP/Stall instructions in the modified code. For example, the calculation of the average of other arrays is carried out in the 5 stage pipeline while waiting for the result to be written to register rather than just stalling to negate the Read After Write (RAW) data dependency hazard.   
  
This greatly improves efficiency. The branch delay and jump stall for the modified code is also reduced to a quarter of the original code as it test the condition for i once after every 4th addition rather than every addition.