Project #2

Simulation of CPU, Cache, Bus, and Memory Datapath

By: Jeff Grindel – A20237004

Subhi Beidas – A20245234

Instructor: Dr. Suresh Borkar

ECE 585-01

Due Date: 4/11/2013

I. Executive Summary

This report goes over what project the group was assigned with, how the group went about designing and implementing the project by making certain assumptions, backing up those assumptions with correct output data. The report finishes with a longer conclusion and all the related code attached.

II. Introduction

Goal

The goal of this project is to build a VHDL simulation of a 32-bit version of the MIPS processor. The developed architecture is to include a CPU, cache, bus, and memory modules that emphasize the simulation of hit/miss and data-path cache scenarios for different types of instructions.

Theoretical Background

Cache: In modern computing systems, accessing data directly from disk or memory is relatively time consuming or *expensive.* This could pose an issue if a set variables stored in memory are extensively required by the computing unit since accessing the respective variables could be time consuming. To deal with this, caches are utilized to minimize this overhead. Caches acts a less expensive intermediate storage block that stores blocks of memory that have been access, been accessed or are going to be accessed be the program. If the requested data is in fact present contained in the cache a ***cache hit*** is identified, this request can be served by simply reading the cache, which is comparatively faster than accessing the memory. Otherwise, if the data block is not present a ***cache miss*** is identified and the data has to be fetched from its respective location in memory. For efficiency, multiple caches are usually implemented to perform difference functions. Widely implemented abstractions usually involve the separation of cache into two main functions, *Instruction Cache and Data Cache.*

Instruction Cache: A stores instruction which helps in reducing the cost of going to memory to fetch instructions. In some other cases it also has other functions, such as branch prediction information. Instructions are only read/brought to the iCache and cannot be modified. So when the I-cache is full and a block of instructions is to be placed into the cache, it can usually over-write anywhere in the cache.

Data Cache is a intermediate storage component that contains the application data that is going to be utilized by the processor. Data is loaded from memory into the data cache. The element needed is then loaded from the cache line into a register and the instruction using this value can operate on it.

Along with different cache functions, there also exist different algorithms or *writing policies* that dictate the interaction between the cache and the memory. In this project's case the chosen writing policy implemented was the ***Write Back with write Allocate*** policy which involves the following:

- Write to main memory whenever a write-hit is performed to the cache
- If a write misses, allocate a line in the cache for the data written.

Group Specific Specifications

Instructions

The MIPS processor is to support the three instruction formats of R, I, and J, along with store word and load word. A table was provided in the project specifications that included all the instructions to be designed

OpCode [31 : 26]	Function Field [5 : 0]	Instruction	Operation
100011	--	lw	lw $s1, 200($t3)
101011	--	sw	sw $s3, 100($t4)
000000	100000	add	add $s3, $t3, $t2
000100	--	beq	beq $s5, $t6, 400
		(Custom set)	

The total set you need to design is the core set as above + a custom set designated for you as follows.
Student ID ending in:

1. BNE, LUI
2. NOR, SLL
3. ADDI, LUI
4. BNE, LUI
5. NOR, LUI
6. ANDI, JR
7. BNE, LUI
8. NOR, LUI
9. ANDI, JR
0. ADDI, LUI

As seen in the table, there is a custom set of instructions that are to be implemented, which is chosen based on the last digit of the student ID'. In this groups case, the 4th set (BNE LUI) was chosen. Another group specific specification was the writing policy, which in our case was set to Write Through with Write Allocate.

Write Strategies:

Both of our digits ended in 4, so we used the following write strategies based on the project 2 specifications.

Write Thru:

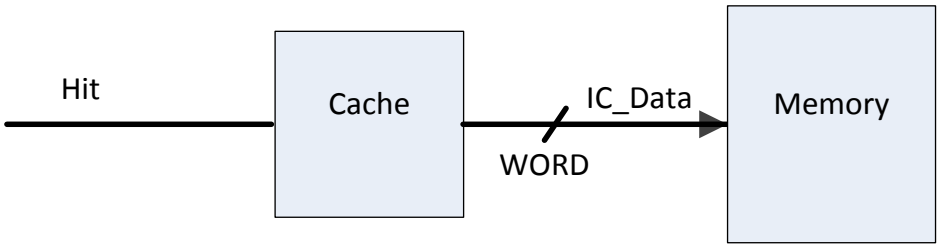


Figure 1: Write Thru

Write Allocate:

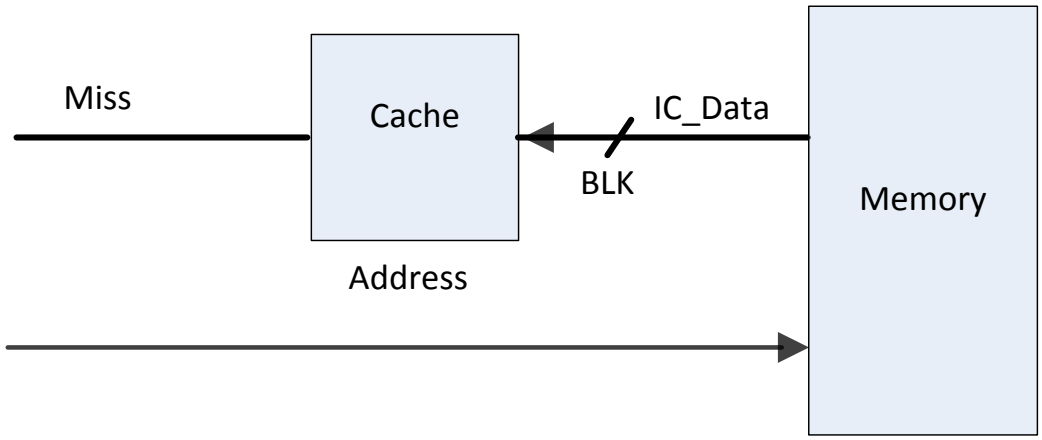


Figure 2: Write Allocate

III. Design

The group decided after lots of discussion to split all of the required pieces of structure into separate modules and then combine them into an overall dataflow. This allowed for each individual component to be thoroughly tested. The overall dataflow can be seen below as well as in the appendix.

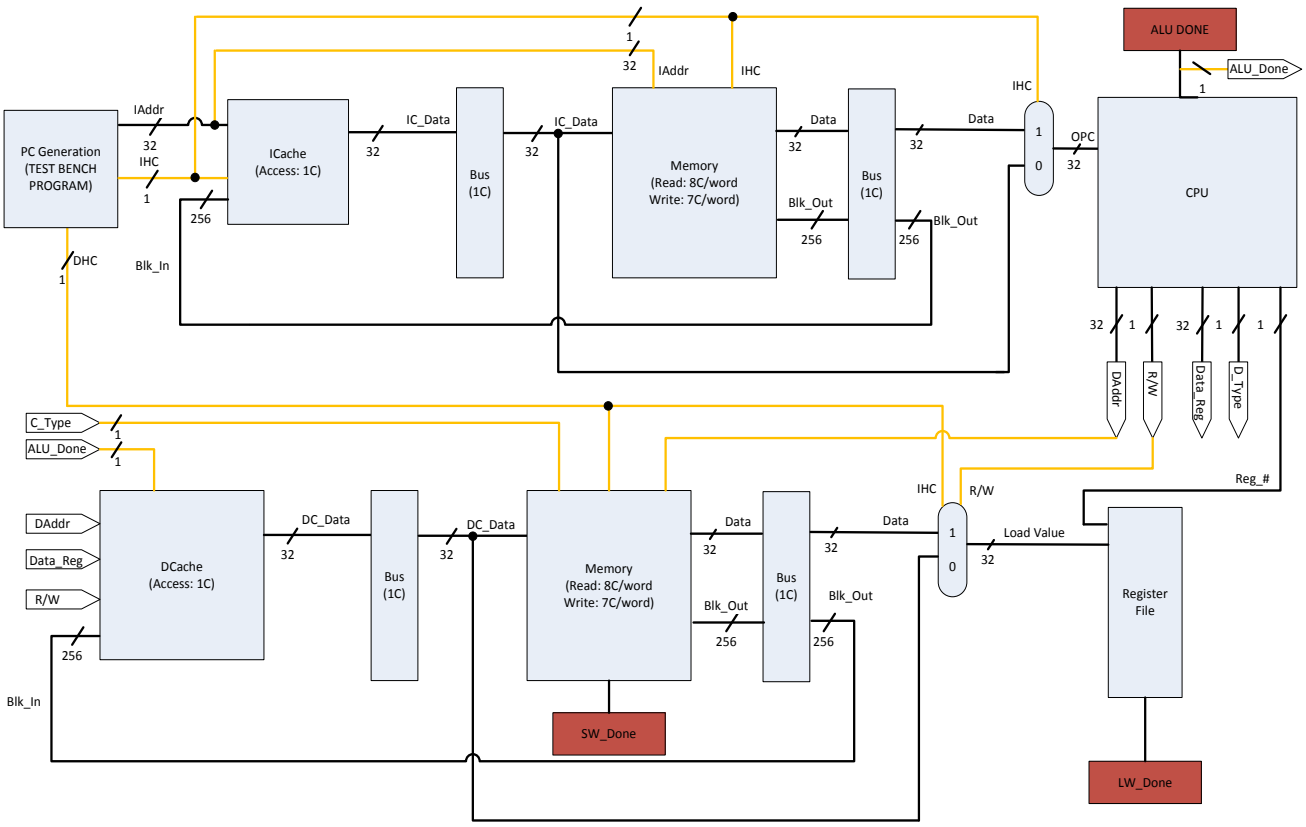


Figure 3: Datapath design of cache/memory system

Based on the figure above, the black lines are mostly data lines, and the orange lines are control inputs for specific block. There are six main blocks where a majority of the functionality takes place.

The ICache block is 256-Byte word-addressable cache (implemented as a 64x32 array, 2048 bits) The cache is directly mapped from memory so the following formula was used to calculate the index of the cache.

$$index = IAddr \ mod \ 64 \qquad (1)$$

This block takes in the Instruction Address (Program Counter) from the test bench program, as well as a flag, IHC, to test the different functionality of either an Icache Hit or Icache Miss. Since we implemented a write through policy, on a cache hit we take the word value from cache and update it to memory. Upon a instruction miss in cache, the data will be fetched from memory. Since the Write Allocate strategy was implemented, a block of memory was put back in the cache to update it. The data is then correct and outputted. The mux on the top is to select either from the memory (ICache Hit) or from the cache after a block update (ICache Miss). The access time for the ICache operation is just 1 cycle (10 ns).

The Bus block is simply to model the bus delay. Since the specifications of the bandwidth of the bus stated, 32 words/cycle, the assumption was made that the bus will delay a full cycle when 32 words or less are put on the bus. Since the most that will put across the bus is a memory block (8 words) this assumption works.

The memory Block is 1024 Bytes of Byte-addressable storage (implemented as a 1024x8 array). The memory then has addresses from 0x0 to 0x3FF. When there is an ICache Hit, a word is brought into the memory to perform a word update, when there is an ICache Miss the Block of memory at the inputted address is outputted and write allocated back to cache to update the values in cache. After the correct instruction is retrieved from either the cache or memory, the Op-Code is inputted into the CPU Block. The memory has a port access time of 5 cycles/word for reads, and an additional memory read time of 3 cycles/word for a total of 8 cycles/word for reading. The memory has a port access time of 3 cycles/word for writes, and an additional memory write time of 4 cycles/word for a total of 7 cycles/word for writing.

The CPU is a simple block, based on the Op-code inputted from the instruction cache/memory process it will determine a couple output signals. The instructions that needed to be implemented are shown below:

Common Name	Register Names
lw $s1, 200 ($t3)	lw R17,200(R11)
sw $s3, 100 ($t4)	sw R19,100(R12)
add $s3, $t3, $t2	add R19,R11,R10
beq $s5, $t6, 400	beq R21,R14,0[400-PC]
bne $s5, $t6, 500	bne R21,R14,0[500-PC]
lui $s6, 40	lui R22,40

Table 1: Instruction Implemented

There are two types of instructions implemented ALU/Branch Instructions and Memory Access functions. The last four on the list above are the ALU/Branch instructions. These only require the use of register values. The Op-codes are determined to match specific instructions. For the beq and bne instructions in the CPU the ALU_DONE flag will be set. For the other two, the register value will be updated with the proper value (add, lui). The Initial values of the register can be seen below.

Register Number	Initial value	Register Number	Initial value
Reg 8	0x00000008	**Reg 17**	0x00000017
Reg 9	0x00000009	**Reg 18**	0x00000018
Reg 10	0x00000010	**Reg 19**	0x00000019
Reg 11	0x00000011	**Reg 20**	0x00000020
Reg 12	0x00000012	**Reg 21**	0x00000021
Reg 13	0x00000013	**Reg 22**	0x00000022
Reg 14	0x00000014	**Reg 23**	0x00000023
Reg 15	0x00000015	**Reg 24**	0x00000024
Reg 16	0x00000016	**Reg 25**	0x00000025

Table 2: Initial Register Values

This will give the following results for the Add and LUI.

Register Names	Results
add R19,R11,R10	R19 <= 0x00000021
lui R22,40	R22<= 0x00280022

Table 3: ALU Instruction Outputs

These two instructions, like the branch instructions will output the ALU_DONE flag to signal that this branch is complete. For Load Word the ALU calculates the Data Address, sets the R_W flag to 1 to signal a read, D_Type flag to signal for the data memory access, and finally the Register number where the data in the memory will be loaded into the register. For Store Word the ALU calculates the Data Address, sets the R_W flag to 0 to signal a write, D_Type flag to signal for the data memory access, and the data in Register 19 (which is initially 0x19).

The DCache datapath will only be run through when there is a data memory access. This is only for the instructions Load word and Store Word, and signaled from the CPU as the D_Type signal. The DCache block is 128-Byte word-addressable cache (implemented as a 32x32 array). The cache is directly mapped from memory so the following formula was used to calculate the index of the cache.

$$index = IAddr \bmod 32 \qquad (2)$$

Once the correct value of the block is reached on a read hit (DCache Hit and Load Word), it will update the word in memory and output that data at the memory address, finally updating the value in the register file. On a read miss (DCache Miss and Load word), it will find the correct value in memory, and write allocate a memory block back to the DCache, updating 8 words in the DCache. On a write hit (DCache Hit and Store Word), the DCache will write Data value output from the ALU to the DCache, then it will be updated in memory. On a write miss (DCache Miss and Store Word), the DCache will store the correct value in memory, and then write allocate a block of memory back to the DCache.

IV. Implementation

To show the implementation of the cache/memory system, code snippets will be shown in respect to their programs. Most everything related specific values in the cache and memory were hardcoded; it made the process of debugging a lot simpler. In order to run through the entire memory sub systems the testbench was written to initialize the Instruction addresses and the Cache Hit types.

The ICache is 256-Bytes of word-addressable memory; this was achieved by using an array. It has inputs of an Instruction Address, the IHC flag, and a Block Input for write allocate. It outputs the data at the instruction address. An array was utilized to define the memory, and all the values were initialized to 0's.

```
type i_array_type is array (0 to 63) of std_logic_vector(31 downto 0);
signal I_Cache : i_array_type := ((others => (others=>'0')));
```

The memory is a direct map to the cache, equation 1 was used to index into the instruction cache array. In this component, if the IHC was asserted, modeling a hit, then the Op-codes would be defined at the address specified. And then the data would be outputted after 1 cycle time to model its access time. If the IHC was not asserted then it would check if a block was being inputted. Upon a miss, one goes to the memory and retrieves the block of memory and write allocates it back into the cache.

The DCache is 128-Bytes of word-addressable memory; this was achieved by using an array. It has inputs of the Data Address, the DHC flag, Data input, block input, a read/write signal. It also has an input to determine if the CPU found a ALU instruction or a memory instruction, this is done using an ALU_Done flag. The Data Cache outputs LW and SW done flags as well as the data at the memory address. Like the ICache, an array was used to simulate the array. This can be seen below.

```
type array_type is array (0 to 31) of std_logic_vector(31 downto 0);
signal D_Cache : array_type := ((others => (others=>'0')));
```

The memory is a direct map to the cache, equation 2 was used to index into the data cache array. In this component the code will only be ran if it is a memory instruction (Ie. ALU_Done is set to 0). This ensures that no data is corrupted in the D-cache dataflow. The Read/Write signal is used to determine if the instruction was a load or store. When R/W is asserted a load instruction is being implemented. On a D-Cache hit, a fake value of x"77777777" is loaded into cache and then outputted to update memory. On a miss, it will go to memory and fetch a block of memory and write allocate it back into the cache. When the Read/Write signal is not asserted, meaning a store instruction is being implemented, on a D-Cache hit; the inputted data value will be stored to the cache and then outputted to update memory. On a D-Cache miss, it goes to memory to get a block to write allocate back to the D-Cache.

The memory module was modeled using an array. It is 1024 Bytes of word addressable. It has all the same inputs, the cache flags, R/W flag, and the address and the data that will update the word upon a hit. It then out puts the data at the memory and/or a block of memory. The array initialization can be seen below.

```
type array_type is array (0 to 1023) of std_logic_vector(7 downto 0);
signal memory : array_type := ((others => (others=>'0')));
```

When there is an instruction hit, the data will be brought in from the cache for the write through strategy. In the following definition C_Type define which cache is being used, 0 for Instruction Cache, and 1 for Data Cache. This is done by the following code.

```
if (IHC = "1" and C_type /= "1") then
            memory(mem_blk) <= Data_In(31 downto 24) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+1)<= Data_In(23 downto 16) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+2) <= Data_In(15 downto 8) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+3) <= Data_In(7 downto 0) after (cycle_time * write_access) +
(cycle_time * write_add);
```

Since in memory has specified access times for read and write. In this case for the write through, it will take 3 Cycles/word for port access then an additional access time of 4 Cycles/word. This leads to 7 cycles/word. For this branch it will take 7 cycles to complete the writing to the memory. This strategy of delaying is used throughout all of the modules to model delay in the circuit. When there is a miss in either the data cache or the instruction cache this module will output a block of memory. It is specified that a block is 8 words, so when there is a read or write the delay times will be multiplied by 8 to account for the 8 separate words.

One of the simpler modules was the modeling the bus. The bus will just output whatever is inputted. Within the module, there is availability to input all the signals that have been described before, however the most important ones are a data word, and a block (8 words). These would just delay one cycle time and then outputted.

The CPU model is a highly simplified version of a typical CPU. In our case, since we are mainly modeling the cache/memory design, we simplified the CPU quite a bit. The CPU will just take in the Op-Code from the instruction cache/memory system. From the Op-code it determines a few key outputs, if it is an ALU instruction, it will set the ALU_Done flag to 1 to signify the ALU operation is done. For the Memory functions, load and store, it will output the respective R/W signal, and the Data Address that was hardcoded (this was done since it was assumed that Register values were initialized and were constant at the each of the instruction). For the Load instruction the following code is used.

```
if (OPC = x"8D7100C8") then    --Load instruction
    report "In load";
    ALU_DONE<="0";
    DAddr<=x"000000D9"; --200 + $11
    R_W<="1";
    Data<="1";
    Data_reg<=x"00000000";
    Reg_Num <= "10001"; --R17
```

One of the last modules that was created was the Register Data component. This simply initializes the register values, and inputs register number from CPU, and data input. For the Add and LUI this is where the Register Values will show being updated. And when the load instruction is inputted, it will load the data value into the specific register.

Once all of the modules were constructed and tested individually, they were combined into two major modules. In order to make the combination of the entire dataflow simple, the modules was split in two. The first one constructed was the ICache_Dataflow that constructed the top half of the Figure showed in the Overall Data flow. This included the ICache, Bus, Memory, a Mux, and finally the CPU. The second one constructed was the DCache_Dataflow that constructed the bottom half of the Figure shown in the Overall

Data flow. This included the DCache, Bus, Memory, a Mux. These two modules were then constructed together along with the register file to obtain the final Testbench program, which we called Cache_Memory_Dataflow. This simple inputs the Instruction address, and the hit flags. It will then out the Done flags, and then finally the data out. When running the program, the register file can also be shown to show the register value changes.

V. Results

To test the implementation of the six instructions, different variables were plugged in the testbench and the respective waveforms were recorded. Below is a sample of some of the results obtained for 2 of the ALU operations (Add,LUI), a load, and a store operation. To ensure that the dChace and the iCache were functioning correctly, different iCache/dCache hit miss combinations were tested across ALU and non-ALU instructions as seen in the following table:

iCahce	dCache	Instruction tested
Hit	*	Add(ALU)
Miss	*	LUI(ALU)
Hit	Hit	LW
Hit	Miss	SW

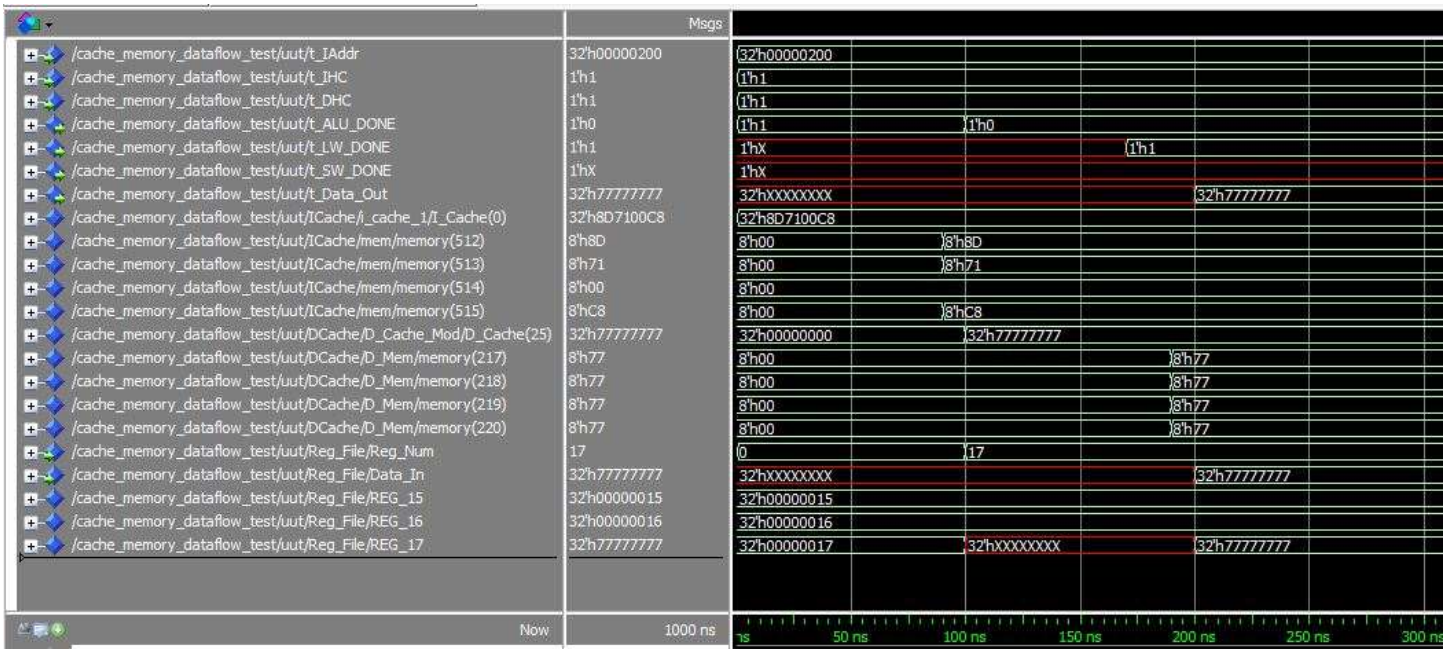
Table 4: Instructions Tested

Load

Instruction Address	Instruction stored	Opcode
0x200	lw $s1, 200 ($t3)	0x8D7100C8

Table 5: Load Instruction Test

Test when iCache hit and dCache hit:
The instruction load word is meant to load the word stored at 200($11) and store it in t3 (Reg 17).The following waveform was obtained by the test bench



The waveform shows the following signals:

- **iAddr (instruction address)**: This input is set to 0x200 which is the hardcoded address for our load instruction
- **IHC (instruction hit flag):** This input is set to 1 to denote a iCache Hit
- **DHC (data hit flag):** This input is set to 1 to denote a dCache Hit
- **LW_DONE:** set to high denoting that CPU has recognized the Load instruction
- **Data_out:** set to 0x77777777 which shows the data stored at the location 200($11)
- **Reg_17:** shows how the data loaded from memory stored in the destination $s1 or Register 17 i.e the success of the simulation

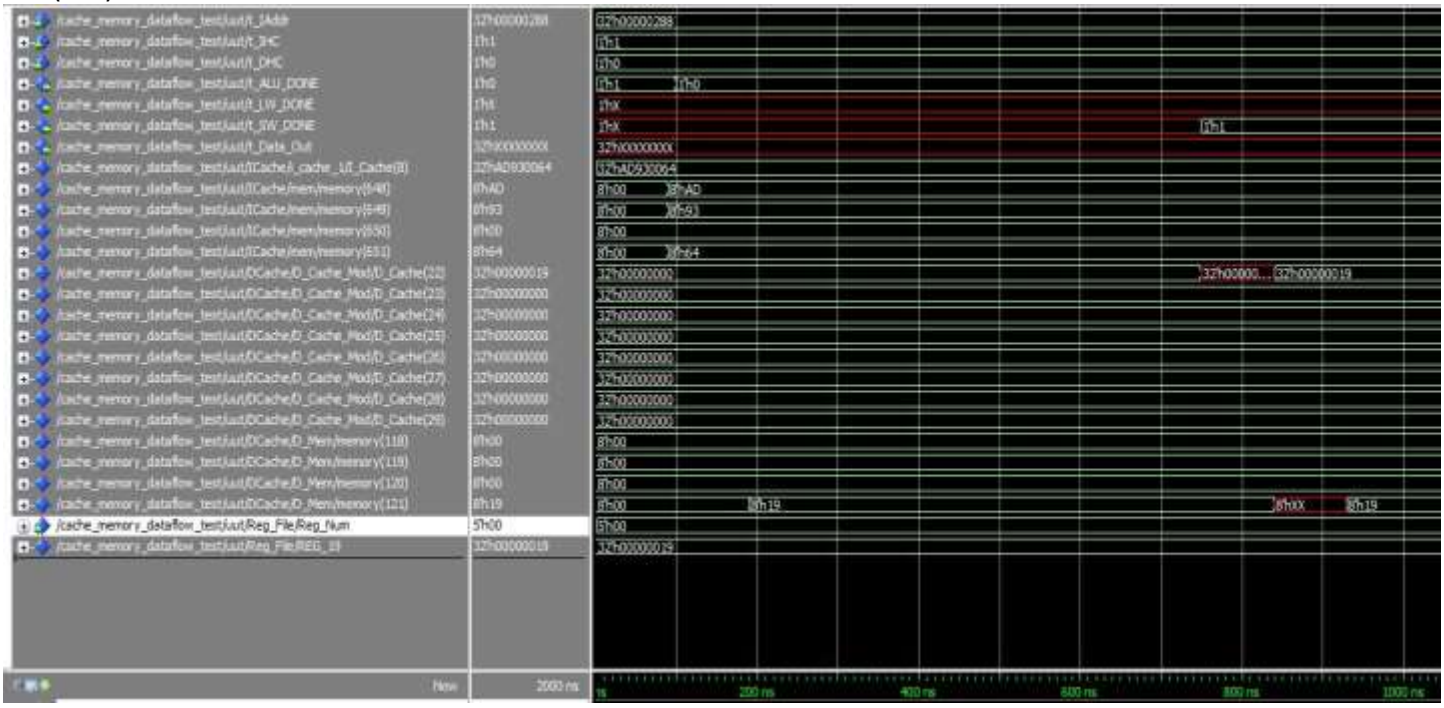
Store

Instruction Address	Instruction stored	Opcode
0x288	sw $s3, 100 ($t4)	0xAD930064

Table 6: Store Instruction Test

Test when iCache hit and dCache miss

The instruction Store word is meant to store the word at $s3(Reg[19]) to the memory location designated by 100($t4).



The waveform shows the following signals:

- **iAddr (instruction address)**: This input is set to 0x288 which is the hardcoded address for our store instruction
- **IHC (instruction hit flag):** This input is set to 1 to denote a iCache Hit
- **DHC (data hit flag):** This input is set to 0 to denote a dCache Miss
- **SW_DONE:** set to high denoting that CPU has recognized it is a store operation
- **Reg_19:** shows the data to be stored to memory stored in the destination memory
- **Memory_121:** The destination memory shows 0x19 (the data from reg 19) stored i.e the success of the simulation

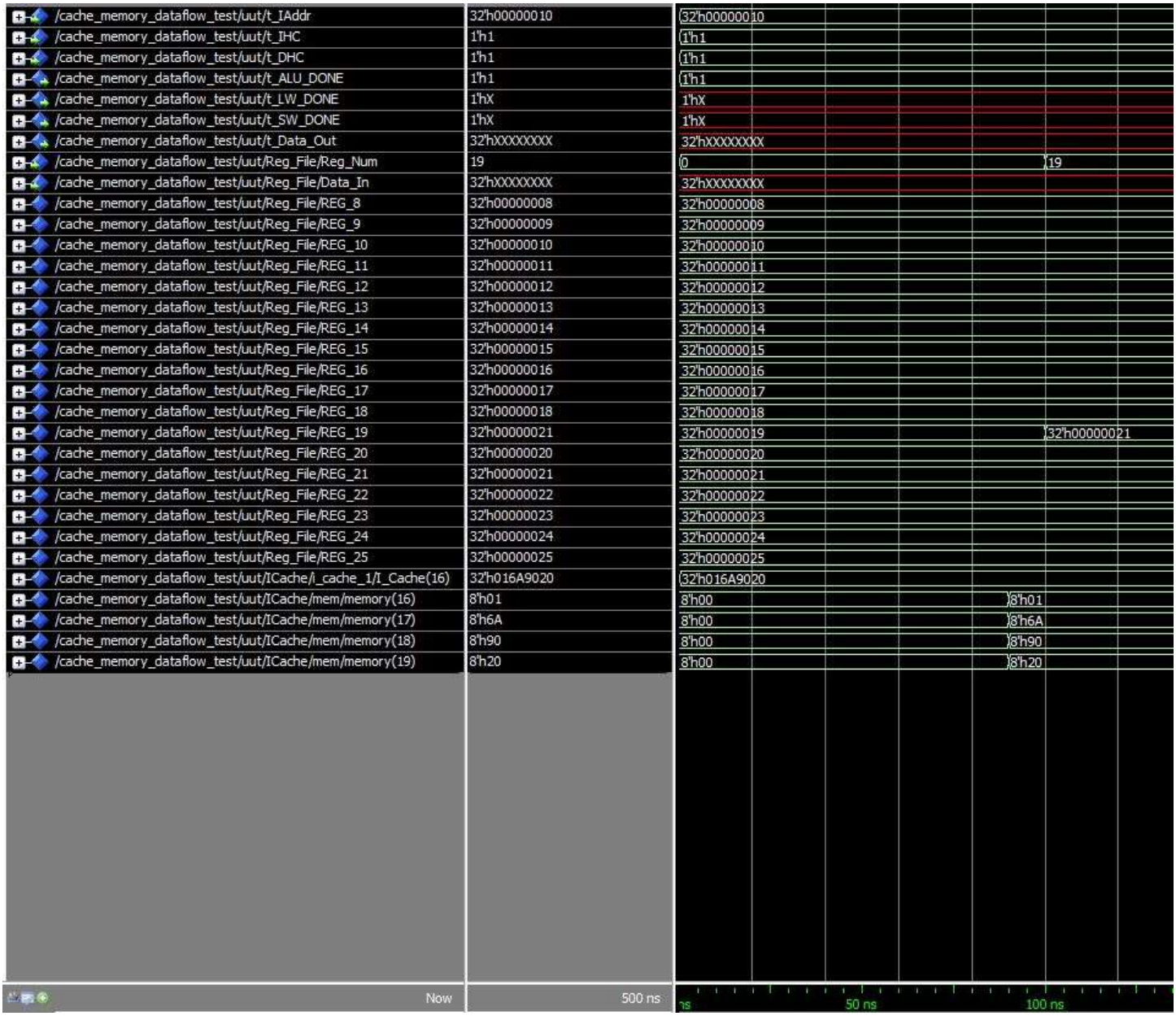
Add

Instruction Address	Instruction stored	Opcode
0x010	add $s3, $t3, $t2	0x016A9020

Table 5: Load Instruction Test

Adds two registers (Reg[10] and Reg[11]) and stores the value in 19. In our module this is an ALU operation and hence should assert the ALU_DONE signal.

Test when iCache hit and dCache hit



The waveform shows the following signals:

- **iAddr (instruction address)**: This input is set to 0x010 which is the hardcoded address for our add instruction
- **IHC (instruction hit flag):** This input is set to 1 to denote a iCache Hit
- **ALU_DONE:** set to 1 denoting that CPU has recognized it is a ALU operation
- **Reg_10:** shows the value stored in register 10 which is set to 0x10
- **Reg_11:** shows the value stored in register 11 which is set to 0x11
- **Reg_19:** shows the value stored in register 19 which was at 0x19 at the beginning of the program and then was set to 0x21 (the summation of Reg_10 and Reg_11) which denotes a successful simulation of the instruction

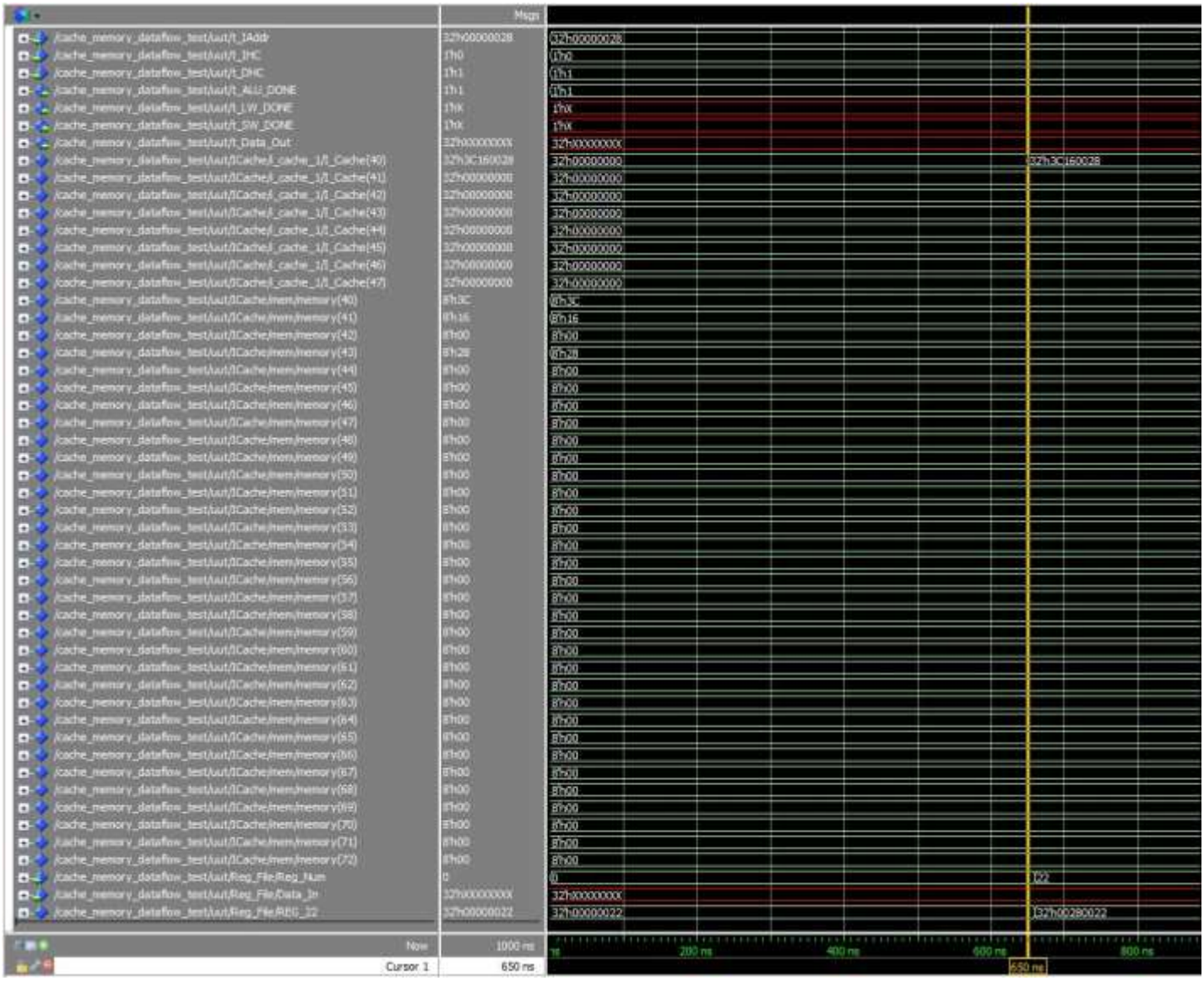
Load upper immediate

Instruction Address	Instruction stored	Opcode
0x028	**lui $s6, 40**	0x3C160028

Table 6: LUI Instruction Test

The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes. The immediate value 40 is shifted 16 bits and stored in $s6 Reg[22]

Test when iCache miss and dCache hit



- **iAddr (instruction address)**: This input is set to 0x028 which is the hardcoded address for our lui instruction
- **IHC (instruction hit flag):** This input is set to 1 to denote a iCache Hit
- **ALU_DONE:** set to 1 denoting that CPU has recognized it is a ALU operation
- **Reg_22:** shows the value stored in register 22 which was at 0x2 at the beginning of the program and then was set to 0x280022 which represent 40(0x28) shifted by 16 bits and appended to the register denotes a successful simulation of the instruction

VI. Discussion

The waveforms obtained are consistent with the expected behavior and thus represent a successful implementation of the modules.

In terms of development, the process went very smoothly and no major bugs were hard to deal with. This could primarily be attributed to the Object Oriented programming approach and testing techniques the team implemented. Writing simple component entities with specific tasks and then writing a test bench for each separate entity has definitely made the debugging process easier. The present code structure is also very scalable; having the separated modules facilitates the accommodation of any code feature addition/improvement in the future.

Even though this is probably outside the scope of the project specification, some nice future work could include deprecating some of the hardcoded instruction opcodes functionality, instead we can have the CPU get the instruction opcode, look it in a dictionary and ultimately be able to actually decode the instruction machine code rather than dealing with the hardcoded hex value.

VII. Conclusions

In conclusion, the developed code was successful in meeting the specifications. With hardcoding everything, the group knew what values would appear, and what each instruction would do. Knowing it all made everything a little bit easier to code. The report shows how each component was set up and how it all came together at the end in the final test bench program. The lab served as a good practical simulation for the expected workflow and ultimately provided the team with a more hands-on insight in respect to the operation of caches.

VIII. Attachments

Each individual program's code will be attached, the individual test benches for the components won't be necessary, since the overall test bench works without issue.

ICache:

```
--I-Cache: 256 Bytes (Word Aaddressable)
--Access time: 1 cycle

library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.all;

entity I_Cache is
   port (IAddr : in std_logic_vector;                          --Instruction address
         IHC : in std_logic_vector(0 downto 0);                --I-Cahche hit flag 1: Hit 0:
Miss
         Blk_In: in std_logic_vector(255 downto 0);            --Block size of 8 words
         I_Cache_Data: out std_logic_vector(31 downto 0));     --Data output of 32-bit
memory
end entity I_Cache;

architecture behave of I_Cache is
    --initialaztion of a memory array of 256 byte (word addressable 32bit data
    type i_array_type is array (0 to 63) of std_logic_vector(31 downto 0);
    signal I_Cache : i_array_type := ((others => (others=>'0')));   --Initialize
everything to 0
    signal temp : integer;

    --signal mem_blk : natural;
    shared variable mem_blk : natural;

    --Address in cache for the instructions
    constant lw_addr : integer := 0; --504-600  --0x200 (512)
    constant sw_addr : integer := 8; --604-700  --0x288 (648)
    constant add_addr : integer := 16; --0-500  --0x010 (16)
    constant beq_addr : integer := 24; --0-500  --0x018 (24)
    constant bne_addr : integer := 32; --0-500  --0x020 (32)
    constant lui_addr : integer := 40; --0-500  --0x028 (40)

    --Cycle Time Constant
    constant cycle_time : time := 10 ns;

begin
    ICache_Proc :process  (IAddr, IHC, Blk_In)
    begin
    mem_blk := to_integer(unsigned(IAddr)) mod 64;
    temp <= mem_blk;
        if(IHC = "1") then
            --Simulation of a I-Cache hit, Init OPC to already be in the cache
            I_Cache(lw_addr) <= x"8D7100C8";
            I_Cache(sw_addr) <= x"AD930064";
            I_Cache(add_addr) <= x"016A9020";
            I_Cache(beq_addr) <= x"12AE0178";
            I_Cache(bne_addr) <= x"16AE01D4";
            I_Cache(lui_addr) <= x"3C160028";
        elsif(IHC = "0") then
            if (Blk_In(255) /= 'U') then    --only does blk replacment when a blk_in is
inputed
```

```
                --Simulation of a I-Cache miss, will write a blk into cache, write
allocate
                I_Cache(mem_blk) <= Blk_In(255 downto 224);
                I_Cache(mem_blk + 1) <= Blk_In(223 downto 192);
                I_Cache(mem_blk + 2) <= Blk_In(191 downto 160);
                I_Cache(mem_blk + 3) <= Blk_In(159 downto 128);
                I_Cache(mem_blk + 4) <= Blk_In(127 downto 96);
                I_Cache(mem_blk + 5) <= Blk_In(95 downto 64);
                I_Cache(mem_blk + 6) <= Blk_In(63 downto 32);
                I_Cache(mem_blk + 7) <= Blk_In(31 downto 0);
            end if;
        end if;
    end process ICache_Proc;

    --Output The opcode
    I_Cache_Data <= I_Cache(to_integer(unsigned(IAddr)) mod 64) after cycle_time when
(IHC = "1") else  --If hit
                    I_Cache(to_integer(unsigned(IAddr)) mod 64) after cycle_time when
(IHC = "0" and Blk_In(255) /= 'U');
                    --Blk_In(255 downto 224) after cycle_time when (IHC = "0" and
Blk_In(255) /= 'U');

end architecture behave;
```

DCache:

```
--D_Cache: 128 Bytes (Word Aaddressable)
--Access time: 1 cycle

library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.all;

entity D_Cache is
   port (DAddr : in std_logic_vector;                    --Data address
         DHC : in std_logic_vector(0 downto 0);          -- 1 bit DCache Flag Input (1 for
hit, 0 for miss, given by testbecnh)
         Data_In : in std_logic_vector(31 downto 0);     --32-bit data in
         Blk_In: in std_logic_vector(255 downto 0);      --Block size of 8 words
         R_W : in std_logic_vector(0 downto 0);          -- 1 for load, 0 for store
         ALU_Done : in std_logic_vector (0 downto 0);    --1 for done, 0 for not done
         LW_Done : out std_logic_vector (0 downto 0);    --1 for done, 0 for not done
         SW_Done : out std_logic_vector (0 downto 0);    --1 for done, 0 for not done
         D_Cache_Data: out std_logic_vector(31 downto 0));--data output of 32-bit memory
end entity D_Cache;

architecture behave of D_Cache is
    --initialaztion of a memory array of 128 byte (word addressable 32bit data
    type array_type is array (0 to 31) of std_logic_vector(31 downto 0);
    signal D_Cache : array_type := ((others => (others=>'0'))); --Initialize everything
to 0
    shared variable mem_blk : natural;

    --Address positions
    constant cycle_time : time := 10 ns;

begin
    DCache_Proc : process (Daddr, DHC, R_W, ALU_Done, Blk_In)
    begin
        mem_blk := to_integer(unsigned(DAddr)) mod 32;
        --Not an alu operation -> SW or LW
        if (ALU_Done = "0") then
            --Load Hit
            if (DHC = "1" and R_W = "1") then
```

```
                --Fake cache inililzation
                D_Cache(mem_blk) <= x"77777777";        --Fake Data at cache address
                LW_Done <= "1";
            --Load Miss
            elsif (DHC = "0" and R_W = "1") then
                if (Blk_In(255) /= 'U') then    --Only does the blk replacment if a Blk
is inputed
                    D_Cache(mem_blk) <= Blk_In(255 downto 224);
                    D_Cache(mem_blk + 1) <= Blk_In(223 downto 192);
                    D_Cache(mem_blk + 2) <= Blk_In(191 downto 160);
                    D_Cache(mem_blk + 3) <= Blk_In(159 downto 128);
                    D_Cache(mem_blk + 4) <= Blk_In(127 downto 96);
                    D_Cache(mem_blk + 5) <= Blk_In(95 downto 64);
                    D_Cache(mem_blk + 6) <= Blk_In(63 downto 32);
                    D_Cache(mem_blk + 7) <= Blk_In(31 downto 0);
                    LW_Done <= "1";
                end if;
            --SW Hit(Address is in cache)
            elsif (DHC = "1" and R_W = "0") then
                D_Cache(mem_blk) <= Data_In;
                SW_Done <= "1";
            elsif (DHC = "0" and R_W = "0") then
                if (Blk_In(255) /= 'U') then --only does the blk replacement if a blk is
inputed
                    D_Cache(mem_blk) <= Blk_In(255 downto 224);
                    D_Cache(mem_blk + 1) <= Blk_In(223 downto 192);
                    D_Cache(mem_blk + 2) <= Blk_In(191 downto 160);
                    D_Cache(mem_blk + 3) <= Blk_In(159 downto 128);
                    D_Cache(mem_blk + 4) <= Blk_In(127 downto 96);
                    D_Cache(mem_blk + 5) <= Blk_In(95 downto 64);
                    D_Cache(mem_blk + 6) <= Blk_In(63 downto 32);
                    D_Cache(mem_blk + 7) <= Blk_In(31 downto 0);
                    SW_Done <= "1";
                end if;
            end if;
        end if;
    end process DCache_Proc;

    --Look into how the memory and cache should be mapped to each other in notes
    D_Cache_Data <= D_Cache(to_integer(unsigned(DAddr)) mod 32) after cycle_time when
(ALU_Done = "0" and DHC = "1" and R_W = "1") else
                    D_Cache(to_integer(unsigned(DAddr)) mod 32) after cycle_time when
(ALU_Done = "0" and DHC = "0" and R_W = "1" and Blk_In(255) /= 'U') else
                    D_Cache(to_integer(unsigned(DAddr)) mod 32) after cycle_time when
(ALU_Done = "0" and DHC = "1" and R_W = "0") else --SW Hit, update mem
                    D_Cache(to_integer(unsigned(DAddr)) mod 32) after cycle_time when
(ALU_Done = "0" and DHC = "0" and R_W = "0" and Blk_In(255) /= 'U') else
                    Data_In after cycle_time when (ALU_Done = "0" and DHC = "0" and R_W =
"0");

end architecture behave;
```

Memory:

```
--Memory: 1024 Bytes (Byte Aaddressable)
--Port Access Time: READ  : 5 cycles/word
--                  WRITE : 3 cycles/word
--
--Addition Time:    READ  : 3 cycles/word
--                  WRITE : 4 cycles/word

library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.all;

entity Memory is
   port (Addr : in std_logic_vector;                    --32-bit Address between 0x0-
0x3FF
         IHC : in std_logic_vector(0 downto 0);         --1 bit ICache Flag Input (1 for
hit, 0 for miss, given by testbecnh)
         DHC : in std_logic_vector(0 downto 0);         --1 bit DCache Flag Input (1 for
hit, 0 for miss)
         R_W : in std_logic_vector(0 downto 0);         --1 for read(Load), 0 for
write(store)
         Data_In : in std_logic_vector(31 downto 0);    --Data input used in SW
Instruction
         C_type : in std_logic_vector(0 downto 0);      --1 for Data cache access, 0 for
Instruction cache access
         LW_Done : out std_logic_vector(0 downto 0);    --Load Word Done Flag
         SW_Done : out std_logic_vector(0 downto 0);    --Store Word Done Flag
         Data_Out: out std_logic_vector(31 downto 0);   --data output of 32-bit memory
         Blk_Out: out std_logic_vector(255 downto 0));  --Block size of 8 words
end entity Memory;

architecture behave of Memory is
    --initialaztion of a memory array of 1024 byte (Byte addressable decimal: 0-1023
(hex: 0x0 -> 0x3FF)
    type array_type is array (0 to 1023) of std_logic_vector(7 downto 0);
    signal memory : array_type := ((others => (others=>'0')));  --Initialize everything
to 0
    shared variable mem_blk : natural;

    signal temp_blk_out: std_logic_vector(255 downto 0);

    --Instruion Positions (addr of the instructions):
    constant lw_addr : integer := 512; --0x200
    constant sw_addr : integer := 648; --0x288
    constant add_addr : integer := 16;  --0x010
    constant beq_addr : integer := 24;  --0x018
    constant bne_addr : integer := 32;  --0x020
    constant lui_addr : integer := 40;  --0x028

    --Cycle Time and Access time constants multipliers
    constant cycle_time : time := 10 ns;
    constant read_access : integer :=  5;
    constant write_access : integer := 3;
    constant read_add : integer := 3;
    constant write_add : integer := 4;

begin
    mem_proc : process (Addr, IHC, DHC, R_W, Data_In, C_type)
    begin
        mem_blk := to_integer(unsigned(Addr));

        --I-Cahce Hit -> Write Thru (Write single word to memory)
        if (IHC = "1" and C_type /= "1") then
```

```
            memory(mem_blk) <= Data_In(31 downto 24) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+1) <= Data_In(23 downto 16) after (cycle_time * write_access)
+ (cycle_time * write_add);
            memory(mem_blk+2) <= Data_In(15 downto 8) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+3) <= Data_In(7 downto 0) after (cycle_time * write_access) +
(cycle_time * write_add);

        --I-Cache Miss -> Write Allocate (Writes block to cache)
        elsif (IHC = "0" and C_type /= "1") then
            --Init of fake instruction memory
            memory(lw_addr) <= x"8D";
            memory(lw_addr+1) <= x"71";
            memory(lw_addr+2) <= x"00";
            memory(lw_addr+3) <= x"C8";

            memory(sw_addr) <= x"AD";
            memory(sw_addr+1) <= x"93";
            memory(sw_addr+2) <= x"00";
            memory(sw_addr+3) <= x"64";

            memory(add_addr) <= x"01";
            memory(add_addr+1) <= x"6A";
            memory(add_addr+2) <= x"90";
            memory(add_addr+3) <= x"20";

            memory(beq_addr) <= x"12";
            memory(beq_addr+1) <= x"AE";
            memory(beq_addr+2) <= x"01";
            memory(beq_addr+3) <= x"78";

            memory(bne_addr) <= x"16";
            memory(bne_addr+1) <= x"AE";
            memory(bne_addr+2) <= x"01";
            memory(bne_addr+3) <= x"D4";

            memory(lui_addr) <= x"3C";
            memory(lui_addr+1) <= x"16";
            memory(lui_addr+2) <= x"00";
            memory(lui_addr+3) <= x"28";

        --D-Cache Hit -> Write Thru (Write signle word to memory)
        --R_W = 1 -> Load Branch (Outputs data at mem)
        elsif (DHC = "1" and R_W = "1" and C_type = "1") then
            memory(mem_blk) <= Data_In(31 downto 24) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+1) <= Data_In(23 downto 16) after (cycle_time * write_access)
+ (cycle_time * write_add);
            memory(mem_blk+2) <= Data_In(15 downto 8) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+3) <= Data_In(7 downto 0) after (cycle_time * write_access) +
(cycle_time * write_add);
            --LW_Done <= "1" after (cycle_time * write_access) + (cycle_time *
write_add);

        --D-Cache Miss -> Write Allocate (Writes block to cache)
        --R_w = 1 -> Load Branch (Outputs blk at mem)
        elsif (DHC = "0" and R_W = "1" and C_type = "1") then
            memory(mem_blk) <= x"EE";
            memory(mem_blk+1) <= x"EE";
            memory(mem_blk+2) <= x"DD";
            memory(mem_blk+3) <= x"DD";
```

```
        --D-Cache Hit -> Write Thru (Write single word to memory)
        --R_W = 0 -> Store Branch
        elsif (DHC = "1" and R_W = "0" and C_type = "1") then
            memory(mem_blk) <= Data_In(31 downto 24) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+1) <= Data_In(23 downto 16) after (cycle_time * write_access)
+ (cycle_time * write_add);
            memory(mem_blk+2) <= Data_In(15 downto 8) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+3) <= Data_In(7 downto 0) after (cycle_time * write_access) +
(cycle_time * write_add);
            --SW_Done <= "1" after (cycle_time * write_access) + (cycle_time *
write_add);

        --D-Cache Miss -> Write Allocate (Writes block to cache)
        --R_W = 0 -> Store Word (Write word to mem, then read blks to update cache)
        elsif (DHC = "0" and R_W = "0" and C_type = "1") then
            memory(mem_blk) <= Data_In(31 downto 24) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+1) <= Data_In(23 downto 16) after (cycle_time * write_access)
+ (cycle_time * write_add);
            memory(mem_blk+2) <= Data_In(15 downto 8) after (cycle_time * write_access) +
(cycle_time * write_add);
            memory(mem_blk+3) <= Data_In(7 downto 0) after (cycle_time * write_access) +
(cycle_time * write_add);
        end if;
    end process mem_proc;

    --Writing the blk into the temp_blk
    temp_blk_out(255 downto 224) <= memory(mem_blk) & memory(mem_blk + 1) &
memory(mem_blk + 2) & memory(mem_blk + 3);
    temp_blk_out(223 downto 192) <= memory(mem_blk + 4) & memory(mem_blk + 5) &
memory(mem_blk + 6) & memory(mem_blk + 7);
    temp_blk_out(191 downto 160) <= memory(mem_blk + 8) & memory(mem_blk + 9) &
memory(mem_blk + 10) & memory(mem_blk + 11);
    temp_blk_out(159 downto 128) <= memory(mem_blk + 12) & memory(mem_blk + 13) &
memory(mem_blk + 14) & memory(mem_blk + 15);
    temp_blk_out(127 downto 96) <= memory(mem_blk + 16) & memory(mem_blk + 17) &
memory(mem_blk + 18) & memory(mem_blk + 19);
    temp_blk_out(95 downto 64) <= memory(mem_blk + 20) & memory(mem_blk + 21) &
memory(mem_blk + 22) & memory(mem_blk + 23);
    temp_blk_out(63 downto 32) <= memory(mem_blk + 24) & memory(mem_blk + 25) &
memory(mem_blk + 26) & memory(mem_blk + 27);
    temp_blk_out(31 downto 0) <= memory(mem_blk + 28) & memory(mem_blk + 29) &
memory(mem_blk + 30) & memory(mem_blk + 31);

    --When IHC = 1, write Data Out
    Data_Out <= Data_In after (cycle_time * write_access) + (cycle_time * write_add) when
(IHC = "1" and C_type /= "1") else
                Data_In after (cycle_time * write_access) + (cycle_time * write_add) when
(DHC = "1" and R_W = "1" and C_type = "1") else
                Data_In after (cycle_time * write_access) + (cycle_time * write_add) when
(DHC = "0" and R_W = "0" and C_type = "1");

    --When IHC = 0, write Blk_Out
    Blk_Out <= temp_blk_out after 8*((cycle_time * read_access) + (cycle_time *
read_add)) when (IHC = "0" and C_type /= "1") else
               temp_blk_out after 8*((cycle_time * read_access) + (cycle_time *
read_add)) when (DHC = "0" and R_W = "1" and C_type = "1") else
               temp_blk_out after 8*((cycle_time * read_access) + (cycle_time *
read_add)) when (DHC = "0" and R_W = "0" and C_type = "1");
```

```
    LW_Done <= "1" after (cycle_time * write_access) + (cycle_time * write_add) when (DHC
= "1" and R_W = "1" and C_type = "1") else
                "1" after (8*((cycle_time * read_access) + (cycle_time * read_add))) when
(DHC = "0" and R_W = "1" and C_type = "1");
    SW_Done <= "1" after (cycle_time * write_access) + (cycle_time * write_add) when (DHC
= "1" and R_W = "0" and C_type = "1") else
                "1" after (8*((cycle_time * read_access) + (cycle_time * read_add))) when
(DHC = "0" and R_W = "0" and C_type = "1") ;
end architecture behave;
```

CPU:

```
--CPU:

library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;

entity CPU is
   port (OPC : in std_logic_vector(31 downto 0);--OP code for instruction
   ALU_DONE: out std_logic_vector (0 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   R_W: out std_logic_vector (0 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   DAddr: out std_logic_vector (31 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   Data: out std_logic_vector (0 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   Data_reg: out std_logic_vector (31 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   Reg_Num: out std_logic_vector (4 downto 0)); --5 bit register number
end entity CPU;

architecture behave of CPU is
begin
    OPC_Proc : process (OPC)
        begin
        -- Check type of instruction
            if (OPC = x"8D7100C8") then   --Load instruction
                report "In load";
                ALU_DONE<="0";
                DAddr<=x"000000D9"; --200 + $11
                R_W<="1";
                Data<="1";
                Data_reg<=x"00000000";
                Reg_Num <= "10001"; --R17
            elsif (OPC = x"AD930064") then -- Store instruction
                report "In Store";
                ALU_DONE<="0";
                DAddr<=x"00000076"; --100 + $12
                R_W<="0";
                Data<="1";
                Data_reg<=x"00000019";
            elsif (OPC = x"016A9020") then --Add
                ALU_DONE<="1";
                DAddr<=x"00000000";
                Data<="0";
                Data_reg<=x"00000000";
                Reg_Num <= "10011";
            elsif (OPC = x"3C160028") then --LUI
                ALU_DONE<="1";
```

```
                DAddr<=x"00000000";
                Data<="0";
                Data_reg<=x"00000000";
                Reg_Num <= "10110";
            else  --ALU instruction
                report "In ALU";
                ALU_DONE<="1";
                DAddr<=x"00000000";
                Data<="0";
                Data_reg<=x"00000000";
                Reg_Num <= "00000";
            end if;

    end process OPC_Proc;
end architecture behave;
```

BUS:

```
--Bus: Used to transfer words and blocks between cahce and memeory.
--Bandwidth of 32words/cycle
--Assume that the bw will never be exceeded and will always take
--up the full cycle

library ieee;
use ieee.std_logic_1164.all;

entity Bus_Model is
   port (Addr : in std_logic_vector (31 downto 0);
         IHC : in std_logic_vector (0 downto 0);
         DHC : in std_logic_vector (0 downto 0);
         R_W : in std_logic_vector (0 downto 0);
         C_Type : in std_logic_vector (0 downto 0);
         Data_In : in std_logic_vector (31 downto 0);
         Blk_In : in std_logic_vector (255 downto 0);

         Addr_Out: out std_logic_vector (31 downto 0);
         IHC_Out: out std_logic_vector (0 downto 0);
         DHC_Out : out std_logic_vector (0 downto 0);
         R_W_Out : out std_logic_vector (0 downto 0);
         C_Type_Out : out std_logic_vector (0 downto 0);
         Data_Out : out std_logic_vector (31 downto 0);
         Blk_Out : out std_logic_vector(255 downto 0));
end entity Bus_Model;

architecture behave of Bus_Model is
    constant cycle_time : time := 10 ns;

begin
    Addr_Out <= Addr after cycle_time;
    IHC_Out <= IHC after cycle_time;
    DHC_Out <= DHC after cycle_time;
    R_W_Out <= R_W after cycle_time;
    C_Type_Out <= C_Type after cycle_time;
    Data_Out <= Data_In after cycle_time;
    Blk_Out <= Blk_In after cycle_time;

end architecture behave;
```

MUX2_32:

```
--MUX FOR 2-input 32-bit Information

library ieee;
use ieee.std_logic_1164.all;

entity Mux2_32 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
          CTRL: in std_logic_vector(0 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Mux2_32;

architecture behavior of Mux2_32 is
begin
    OUTPUT <= ONE when (CTRL = "1") else        --I-TYPE, Outputs IMM
              ZERO;
end architecture behavior;
```

MUX22_32:

```
--MUX FOR 2-input 32-bit Information with 2x1-bit control

library ieee;
use ieee.std_logic_1164.all;

entity Mux22_32 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
          CTRL1: in std_logic_vector(0 downto 0);
          CTRL2: in std_logic_vector (0 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Mux22_32;

architecture behavior of Mux22_32 is
begin
    OUTPUT <= ONE when (CTRL1 = "1" and CTRL2 = "1") else       --I-TYPE, Outputs IMM
              ZERO when (CTRL1 = "0" and CTRL2 = "1");
end architecture behavior;
```

Register_Data:

```
--Register_Data

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Register_Data is
   port (Reg_Num: in std_logic_vector(4 downto 0);       --5-bit Read Reg. 1
         Data_In: in std_logic_vector(31 downto 0));
end entity Register_Data;

architecture behave of Register_Data is

--Changing Vectors
signal REG_8: std_logic_vector(31 downto 0) := x"00000008";
signal REG_9: std_logic_vector(31 downto 0) := x"00000009";
signal REG_10: std_logic_vector(31 downto 0) := x"00000010";
signal REG_11: std_logic_vector(31 downto 0) := x"00000011";
signal REG_12: std_logic_vector(31 downto 0) := x"00000012";
signal REG_13: std_logic_vector(31 downto 0) := x"00000013";
signal REG_14: std_logic_vector(31 downto 0) := x"00000014";
signal REG_15: std_logic_vector(31 downto 0) := x"00000015";
signal REG_16: std_logic_vector(31 downto 0) := x"00000016";
signal REG_17: std_logic_vector(31 downto 0) := x"00000017";
signal REG_18: std_logic_vector(31 downto 0) := x"00000018";
signal REG_19: std_logic_vector(31 downto 0) := x"00000019";
signal REG_20: std_logic_vector(31 downto 0) := x"00000020";
signal REG_21: std_logic_vector(31 downto 0) := x"00000021";
signal REG_22: std_logic_vector(31 downto 0) := x"00000022";
signal REG_23: std_logic_vector(31 downto 0) := x"00000023";
signal REG_24: std_logic_vector(31 downto 0) := x"00000024";
signal REG_25: std_logic_vector(31 downto 0) := x"00000025";

begin
    Reg_19 <= x"00000021" when Reg_Num = "10011"; --simulated addition
    Reg_22 <= x"00280022" when Reg_Num = "10110"; --simulated lui
    Reg_17 <= Data_In when Reg_num = "10001"; --sumulates lw
end architecture behave;
```

ICache_Dataflow:

```
--BETA Class for the instruction cache,mem and CPU process
--NOTE:ADD bus

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity ICache_Dataflow is
  port (aIAddr : in std_logic_vector(31 downto 0);
        aIHC : in std_logic_vector(0 downto 0); --I-Cahche hit flag 1: Hit 0: Miss
        aALU_DONE: out std_logic_vector (0 downto 0);
        aR_W: out std_logic_vector (0 downto 0);
        aDAddr: out std_logic_vector (31 downto 0);
        aData: out std_logic_vector (0 downto 0);
        aData_reg: out std_logic_vector (31 downto 0);
        aReg_Num: out std_logic_vector (4 downto 0)); --5 bit register number
end ICache_Dataflow;

architecture behave of ICache_Dataflow is

COMPONENT I_Cache is
   port (IAddr : in std_logic_vector; -- 0x0 - 0x40
         IHC : in std_logic_vector(0 downto 0); --I-Cahche hit flag 1: Hit 0: Miss
         Blk_In: in std_logic_vector(255 downto 0); --Block size of 8 words
         I_Cache_Data: out std_logic_vector(31 downto 0));  --data output of 32-bit
memory
end COMPONENT;

COMPONENT Mux2_32 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
          CTRL: in std_logic_vector(0 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end COMPONENT;

COMPONENT Memory is
   port (Addr : in std_logic_vector;            --32-bit Address between 0x0-0x3FF
         IHC : in std_logic_vector(0 downto 0); -- 1 bit ICache Flag Input (1 for hit, 0
for miss, given by testbecnh)
         DHC : in std_logic_vector(0 downto 0); -- 1 bit DCache Flag Input (1 for hit, 0
for miss)
         R_W : in std_logic_vector(0 downto 0);     --1 for read(Load), 0 for
write(store)
         Data_In : in std_logic_vector(31 downto 0);-- Data input used in SW Instruction
         C_type : in std_logic_vector(0 downto 0); --1 for Data cache access, 0 for
Instruction cache access
         LW_Done : out std_logic_vector(0 downto 0);
         SW_Done : out std_logic_vector(0 downto 0);

        Data_Out: out std_logic_vector(31 downto 0);    --data output of 32-bit memory
        Blk_Out: out std_logic_vector(255 downto 0));   --Block size of 8 words
end COMPONENT;

COMPONENT CPU is
   port (OPC : in std_logic_vector(31 downto 0);--OP code for instruction
   ALU_DONE: out std_logic_vector (0 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   R_W: out std_logic_vector (0 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   DAddr: out std_logic_vector (31 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
```

```
   Data: out std_logic_vector (0 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   Data_reg: out std_logic_vector (31 downto 0); --2 bit Instruction type flag (0==Load)
(1==Store) (2==Alu)
   Reg_Num: out std_logic_vector (4 downto 0)); --5 bit register number

end COMPONENT;

COMPONENT Bus_Model is
   port (Addr : in std_logic_vector;
         IHC : in std_logic_vector (0 downto 0);
         DHC : in std_logic_vector (0 downto 0);
         R_W : in std_logic_vector (0 downto 0);
         C_Type : in std_logic_vector (0 downto 0);
         Data_In : in std_logic_vector (31 downto 0);
         Blk_In : in std_logic_vector (255 downto 0);

         Addr_Out: out std_logic_vector;
         IHC_Out: out std_logic_vector (0 downto 0);
         DHC_Out : out std_logic_vector (0 downto 0);
         R_W_Out : out std_logic_vector (0 downto 0);
         C_Type_Out : out std_logic_vector (0 downto 0);
         Data_Out : out std_logic_vector (31 downto 0);
         Blk_Out : out std_logic_vector(255 downto 0));
end COMPONENT;


-------------------------------------------------First Cache Signals----------------------
--------------------------------
--Declaration of Inputs
signal ICACHE_I_IAddr : std_logic_vector (7 downto 0);
signal ICACHE_I_IHC : std_logic_vector(0 downto 0); --I-Cahche hit flag
signal ICACHE_I_Blk_In: std_logic_vector(255 downto 0);
--Declaration of Outputs
signal ICACHE_I_I_Cache_Data: std_logic_vector(31 downto 0);
-------------------------------------------------Second Cache Signals---------------------
--------------------------------
--Declaration of Inputs
signal ICACHE_II_IAddr : std_logic_vector (7 downto 0);
signal ICACHE_II_IHC : std_logic_vector(0 downto 0); --I-Cahche hit flag
signal ICACHE_II_Blk_In: std_logic_vector(255 downto 0); --1 for read, 0 for write
--Declaration of Outputs
signal ICACHE_II_cache_Data: std_logic_vector(31 downto 0);
-------------------------------------------------Memory Signals---------------------------
--------------------------------
--Declaration of test signal Inputs
signal MEM_Addr : std_logic_vector(31 downto 0);
signal MEM_IHC : std_logic_vector(0 downto 0);
signal MEM_DHC : std_logic_vector(0 downto 0);
signal MEM_R_W : std_logic_vector(0 downto 0); --1 for read, 0 for write
signal MEM_Data_In : std_logic_vector(31 downto 0);
signal MEM_C_type : std_logic_vector(0 downto 0);
--Declaration of test signal Outputs
signal MEM_LW_Done : std_logic_vector(0 downto 0);
signal MEM_SW_Done : std_logic_vector(0 downto 0);
signal  MEM_Data_Out: std_logic_vector(31 downto 0);    --data output of 32-bit MEM
signal  MEM_Blk_Out: std_logic_vector(255 downto 0);    --Block size of 8 words
-------------------------------------------------Mux Signals------------------------------
--------------------------------
--Declaration of test signal Inputs
signal MUX_ZERO: std_logic_vector(31 downto 0);
signal MUX_ONE: std_logic_vector(31 downto 0);
signal MUX_CTRL: std_logic_vector(0 downto 0);
```

```
--Declaration of test signal Outputs
signal MUX_OUTPUT: std_logic_vector(31 downto 0);
-------------------------------------------CPU Signals-----------------------------
---------------------------------
--Declaration of test signal Inputs
signal CPU_OPC :  std_logic_vector(31 downto 0);
--Declaration of test signal Outputs
signal CPU_ALU_DONE:  std_logic_vector (0 downto 0);
signal CPU_R_W:  std_logic_vector (0 downto 0);
signal CPU_DAddr:  std_logic_vector (31 downto 0);
signal CPU_Data:  std_logic_vector (0 downto 0);
signal CPU_Data_reg: std_logic_vector (31 downto 0);

signal T_Addr_Out : std_logic_vector(31 downto 0);
signal T_Data_out : std_logic_vector(31 downto 0);
signal T_Data_out2 : std_logic_vector(31 downto 0);
signal T_Data_out3 : std_logic_vector(31 downto 0);
signal T_Data_out4 : std_logic_vector(31 downto 0);
signal T_Blk_Out2 : std_logic_vector(255 downto 0);

begin
    i_cache_1: I_Cache PORT MAP (
            IAddr=>aIAddr,
            IHC =>aIHC,
            Blk_In =>T_Blk_Out2, --in (Undriven for first cache)
            I_Cache_Data =>T_Data_out);

    Bus_1 : Bus_Model port map (Addr => aIAddr,
                            IHC => aIHC,
                            DHC =>  "U",
                            R_W =>  "U",
                            C_Type => "0",
                            Data_In => T_Data_out,
                            Blk_In =>  MEM_Blk_Out,
                            Addr_Out =>T_Addr_Out,
                            Data_Out =>  T_Data_Out2);

    mem: Memory PORT MAP (
            Addr => aIAddr,--in
            IHC => aIHC,   --in
            DHC => "U",        --in (Undriven for Instruction flow)
            R_W => "U",        --in (Undriven for Instruction flow)
            Data_In => T_Data_Out2,--in
            C_type => "0",  --in   (instruction)
            --LW_Done => "0",   --(Undriven for Instruction flow)
            --SW_Done => "0",   --(Undriven for Instruction flow)
            Data_Out => T_Data_out3, --(Undriven for Instruction flow)
            Blk_Out => MEM_Blk_Out);

    Bus_2 : Bus_Model port map (Addr => aIAddr,
                          IHC => aIHC,
                          DHC =>  "U",
                          R_W =>  "U",
                          C_Type => "0",
                          Data_In => T_Data_out3,
                          Blk_In =>  MEM_Blk_Out,
                          Addr_Out =>T_Addr_Out,
                          Data_Out =>  T_Data_Out4,
                          Blk_Out => T_Blk_Out2);

    -- i_cache_2: I_Cache PORT MAP (
            -- IAddr=>aIAddr,--in
            -- IHC => aIHC,--in
```

```
            -- Blk_In => MEM_Blk_Out,--in
            -- I_Cache_Data =>ICACHE_II_Cache_Data);

    mux: Mux2_32 PORT MAP (
            ZERO => T_Data_out, --in
            ONE => T_Data_Out4,--in
            CTRL => aIHC,--in
            OUTPUT => MUX_OUTPUT);

  cpu_uut: CPU PORT MAP (
            OPC => MUX_OUTPUT, --in
            ALU_DONE => aALU_DONE,
            R_W => aR_W,
            DAddr => aDAddr,
            Data => aData,
            Data_reg => aData_reg,
            Reg_Num=>aReg_Num);



end architecture behave;
```

DCache Dataflow:

```
--Combination of DCache, Mem

library ieee;
use ieee.std_logic_1164.all;

entity DCache_Dataflow is
    port (aDAddr : in std_logic_vector (31 downto 0);
          aData : in std_logic_vector (31 downto 0);
          aBlk : in std_logic_vector (255 downto 0);
          aALU_Done : in std_logic_vector (0 downto 0);
          aR_W : in std_logic_vector (0 downto 0);
          aDHC : in std_logic_vector (0 downto 0);
          aType : in std_logic_vector (0 downto 0);

          aOut : out std_logic_vector (31 downto 0);
          aLW_Done : out std_logic_vector (0 downto 0);
          aSW_Done : out std_logic_vector (0 downto 0));
end entity DCache_Dataflow;

architecture behave of DCache_Dataflow is



--Component declaration
COMPONENT D_Cache is
   port (DAddr : in std_logic_vector;   --Data address
         DHC : in std_logic_vector(0 downto 0); -- 1 bit DCache Flag Input (1 for hit, 0
for miss, given by testbecnh)
         Data_In : in std_logic_vector(31 downto 0); --32-bit data in
         Blk_In: in std_logic_vector(255 downto 0); --Block size of 8 words
         R_W : in std_logic_vector(0 downto 0); -- 1 for load, 0 for store
         ALU_Done : in std_logic_vector (0 downto 0); --1 for done, 0 for not done
         LW_Done : out std_logic_vector (0 downto 0); --1 for done, 0 for not done
         SW_Done : out std_logic_vector (0 downto 0); --1 for done, 0 for not done
         D_Cache_Data: out std_logic_vector(31 downto 0));  --data output of 32-bit
memory
end COMPONENT;

COMPONENT Bus_Model is
   port (Addr : in std_logic_vector;
```

```
          IHC : in std_logic_vector (0 downto 0);
          DHC : in std_logic_vector (0 downto 0);
          R_W : in std_logic_vector (0 downto 0);
          C_Type : in std_logic_vector (0 downto 0);
          Data_In : in std_logic_vector (31 downto 0);
          Blk_In : in std_logic_vector (255 downto 0);

          Addr_Out: out std_logic_vector;
          IHC_Out: out std_logic_vector (0 downto 0);
          DHC_Out : out std_logic_vector (0 downto 0);
          R_W_Out : out std_logic_vector (0 downto 0);
          C_Type_Out : out std_logic_vector (0 downto 0);
          Data_Out : out std_logic_vector (31 downto 0);
          Blk_Out : out std_logic_vector(255 downto 0));
end COMPONENT;

COMPONENT Memory is
   port (Addr : in std_logic_vector;            --32-bit Address between 0x0-0x3FF
          IHC : in std_logic_vector(0 downto 0); -- 1 bit ICache Flag Input (1 for hit, 0
for miss, given by testbecnh)
          DHC : in std_logic_vector(0 downto 0); -- 1 bit DCache Flag Input (1 for hit, 0
for miss)
          R_W : in std_logic_vector(0 downto 0);     --1 for read(Load), 0 for
write(store)
          Data_In : in std_logic_vector(31 downto 0);-- Data input used in SW Instruction
          C_type : in std_logic_vector(0 downto 0); --1 for Data cache access, 0 for
Instruction cache access
          LW_Done : out std_logic_vector(0 downto 0);
          SW_Done : out std_logic_vector(0 downto 0);
         Data_Out: out std_logic_vector(31 downto 0);    --data output of 32-bit memory
         Blk_Out: out std_logic_vector(255 downto 0));   --Block size of 8 words

end COMPONENT;

COMPONENT Mux22_32 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
          CTRL1: in std_logic_vector(0 downto 0);
          CTRL2: in std_logic_vector(0 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end COMPONENT;

--Temp signals used as connections between components
signal T_Data_In : std_logic_vector (31 downto 0);
signal T_Blk_In : std_logic_vector (255 downto 0);
signal T_Data_Out : std_logic_vector (31 downto 0);
signal T_Data_Out2 : std_logic_vector (31 downto 0);
signal T_Data_Out3 : std_logic_vector (31 downto 0);
signal T_Data_Out4 : std_logic_vector (31 downto 0);
signal T_Blk_Out : std_logic_vector (255 downto 0);
signal T_Blk_Out2 : std_logic_vector (255 downto 0);
signal T_Addr_Out : std_logic_vector(31 downto 0);
begin

D_Cache_Mod : D_Cache port map (DAddr => aDAddr,
                                DHC =>  aDHC,
                                Data_In => aData,
                                Blk_In => T_Blk_Out2,
                                R_W => aR_W,
                                ALU_Done => aALU_Done,
                                LW_Done => aLW_Done,
                                SW_Done => aSW_Done,
                                D_Cache_Data => T_Data_Out);
```

```
Bus_1 : Bus_Model port map (Addr => aDAddr,
                            IHC => "U",
                            DHC =>  aDHC,
                            R_W =>  aR_W,
                            C_Type => aType,
                            Data_In => T_Data_Out,
                            Blk_In =>  aBlk,
                            Addr_Out =>T_Addr_Out,
                            Data_Out =>  T_Data_Out2);

D_Mem : Memory port map (Addr => aDAddr,
                         IHC =>  "U",
                         DHC => aDHC,
                         R_W => aR_W,
                         Data_In => T_Data_Out2,
                         C_type => aType,
                         LW_Done => aLW_Done,
                         SW_Done => aSW_Done,
                         Data_Out => T_Data_Out3,
                         Blk_Out => T_Blk_Out);

Bus_2 : Bus_Model port map (Addr => aDAddr,
                            IHC => "U",
                            DHC =>  aDHC,
                            R_W =>  aR_W,
                            C_Type => aType,
                            Data_In => T_Data_Out3,
                            Blk_In =>  T_Blk_Out,
                            Addr_Out =>T_Addr_Out,
                            Data_Out =>  T_Data_Out4,
                            Blk_Out => T_Blk_Out2);

Mux_32 : Mux22_32 port map (ZERO =>     T_Data_Out,
                           ONE => T_Data_Out4,
                           CTRL1 => aDHC,
                           CTRL2 => aR_W,
                           OUTPUT => aOut);


end architecture behave;
```

Cache_Memory_Dataflow:

```
--BETA Class for the instruction cache,mem and CPU process
--NOTE:ADD bus

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity Cache_Memory_Dataflow is
  port (t_IAddr : in std_logic_vector(31 downto 0);
        t_IHC : in std_logic_vector(0 downto 0); --I-Cahche hit flag 1: Hit 0: Miss
        t_DHC : in std_logic_vector(0 downto 0);
        t_ALU_DONE: out std_logic_vector (0 downto 0);
        t_LW_DONE: out std_logic_vector (0 downto 0);
        t_SW_DONE: out std_logic_vector (0 downto 0);
        t_Data_Out: out std_logic_vector (31 downto 0)); --5 bit register number
end Cache_Memory_Dataflow;

architecture behave of Cache_Memory_Dataflow is

COMPONENT ICache_Dataflow is
      port (aIAddr : in std_logic_vector(31 downto 0);
        aIHC : in std_logic_vector(0 downto 0); --I-Cahche hit flag 1: Hit 0: Miss
        aALU_DONE: out std_logic_vector (0 downto 0);
        aR_W: out std_logic_vector (0 downto 0);
        aDAddr: out std_logic_vector (31 downto 0);
        aData: out std_logic_vector (0 downto 0);
        aData_reg: out std_logic_vector (31 downto 0);
        aReg_Num: out std_logic_vector (4 downto 0)); --5 bit register number
end component;

COMPONENT DCache_Dataflow is
    port (aDAddr : in std_logic_vector (31 downto 0);
          aData : in std_logic_vector (31 downto 0);
          aBlk : in std_logic_vector (255 downto 0);
          aALU_Done : in std_logic_vector (0 downto 0);
          aR_W : in std_logic_vector (0 downto 0);
          aDHC : in std_logic_vector (0 downto 0);
          aType : in std_logic_vector (0 downto 0);

          aOut : out std_logic_vector (31 downto 0);
          aLW_Done : out std_logic_vector (0 downto 0);
          aSW_Done : out std_logic_vector (0 downto 0));
end component;

COMPONENT Register_Data is
   port (Reg_Num: in std_logic_vector (4 downto 0);     --5-bit Read Reg. 1
         Data_In: in std_logic_vector (31 downto 0));
end component;

signal temp_rw : std_logic_vector (0 downto 0);
signal temp_DAddr : std_logic_vector (31 downto 0);
signal temp_C_Type : std_logic_vector (0 downto 0);
signal temp_CPU_Data : std_logic_vector (31 downto 0);
signal temp_Reg_Num : std_logic_vector (4 downto 0);
signal not_used_in_this_test : std_logic_vector (255 downto 0);
signal temp_ALU_DONE : std_logic_vector (0 downto 0);
signal temp_Data_Out : std_logic_vector (31 downto 0);
begin
    ICache : ICache_Dataflow port map(aIAddr  => t_IAddr,
                                      aIHC  => t_IHC,
```

```
                                        aALU_DONE => temp_ALU_DONE,
                                        aR_W => temp_rw,
                                        aDAddr => temp_DAddr,
                                        aData => temp_C_Type,
                                        aData_reg => temp_CPU_Data,
                                        aReg_Num =>  temp_Reg_Num);

  DCache : DCache_Dataflow port map(aDAddr => temp_DAddr,
                                    aData =>  temp_CPU_Data,
                                    aBlk =>  not_used_in_this_test,
                                    aALU_Done => temp_ALU_DONE,
                                    aR_W =>  temp_rw,
                                    aDHC => t_DHC,
                                    aType => temp_C_Type,

                                    aOut  =>  temp_Data_Out,
                                    aLW_Done  => t_LW_DONE,
                                    aSW_Done  => t_SW_DONE);

    Reg_File : Register_Data port map(Reg_Num => temp_Reg_Num,
                                      Data_In => temp_Data_Out);

    t_ALU_DONE <= temp_ALU_DONE;
    t_Data_Out <= temp_Data_Out;
end architecture behave;
```

Cache_Memory_Dataflow_Test (Overall Testbench Program):

```
library ieee;
use ieee.std_logic_1164.all;

entity Cache_Memory_Dataflow_Test is
end entity;

architecture behave of Cache_Memory_Dataflow_Test  is


 COMPONENT Cache_Memory_Dataflow is
  port (t_IAddr : in std_logic_vector(31 downto 0);
        t_IHC : in std_logic_vector(0 downto 0); --I-Cahche hit flag 1: Hit 0: Miss
        t_DHC : in std_logic_vector(0 downto 0);
        t_ALU_DONE: out std_logic_vector (0 downto 0);
        t_LW_DONE: out std_logic_vector (0 downto 0);
        t_SW_DONE: out std_logic_vector (0 downto 0);
        t_Data_Out: out std_logic_vector (31 downto 0)); --5 bit register number
end COMPONENT;

 --Declaration of test signal Inputs
signal TB_aIAddr :  std_logic_vector(31 downto 0);
signal TB_aIHC :  std_logic_vector(0 downto 0); --I-Cahche hit flag 1: Hit 0: Miss
signal TB_aDHC :  std_logic_vector(0 downto 0); --D-Cahche hit flag 1: Hit 0: Miss
 --Declaration of test signal Outputs
signal TB_ALU_DONE:  std_logic_vector (0 downto 0);
signal TB_LW_DONE: std_logic_vector (0 downto 0);
signal TB_SW_DONE: std_logic_vector (0 downto 0);
signal TB_Data_Out:  std_logic_vector (31 downto 0);


begin
    uut: Cache_Memory_Dataflow PORT MAP (
            t_IAddr=>TB_aIAddr,
            t_IHC => TB_aIHC,
            t_DHC => TB_aDHC,
            t_ALU_DONE=> TB_ALU_DONE,
```

```
            t_LW_DONE=> TB_LW_DONE,
            t_SW_DONE=> TB_SW_DONE,
            t_Data_Out=>TB_Data_Out);

    aAddr_Gen : process
    begin
        TB_aIAddr <= --x"00000200" after 0 ns;      --Load Address
                      x"00000288" after 0 ns;       --Store Address
                    --  x"00000010" after 0 ns;         --Add Address
                   -- x"00000018" after 0 ns;      --BEQ Address
                   -- x"00000020" after 0 ns;      --BNE Address
                    -- x"00000028" after 0 ns;     --LUI Address

    wait;
    end process aAddr_Gen;

    aIHC_Gen : process
    begin
        TB_aIHC <= "1" after 0 ns;  --ICache Hit
                  --"0" after 0 ns;  --ICache Miss
    wait;
    end process aIHC_Gen;

    aDHC_Gen : process
    begin
        TB_aDHC <= --"1" after 0 ns;    --DCache Hit
                  "0" after 0 ns;    --DCache Miss
    wait;
    end process aDHC_Gen;



end architecture behave;
```