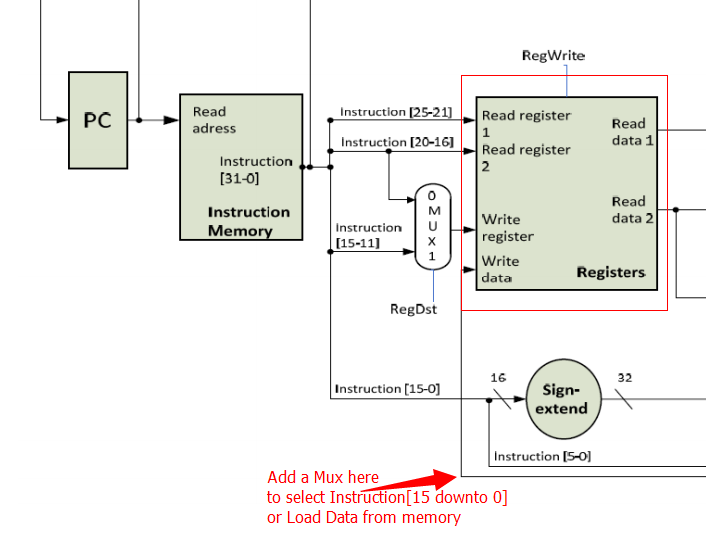
**LUI -- *Load upper immediate implementation***

|  |  |
| --- | --- |
| Description: | The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes. |
| Operation: | $t = (imm << 16); advance\_pc (4); |
| Syntax: | lui $t, imm |
| Encoding: | 0011 11-- ---t tttt iiii iiii iiii iiii |

“Lui” datapath



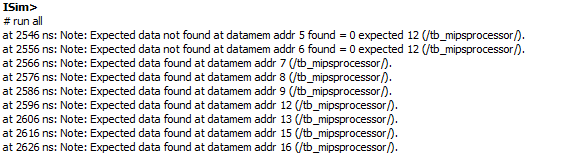
Since this instruction only accesses instruction register and register file, no dmem or alu included, we add a mux to choose the regfile writedata either from new instant value or load value from dmem.

This instruction only affects IF, ID, and WB cycles of the pipeline. At the decode step, RegDst = ‘1’; RegWriteSrc = ‘1’, at the WB step RegWrite = ‘1’.

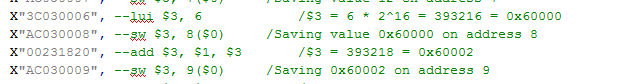
Code

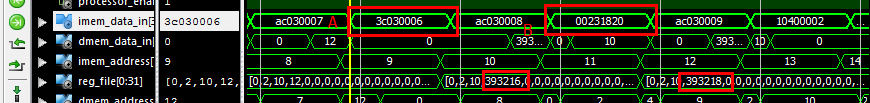
|  |
| --- |
|  |

Testbench running results:



Let’s look into the waveform to verify the execution details of following instructions generated by the testbench





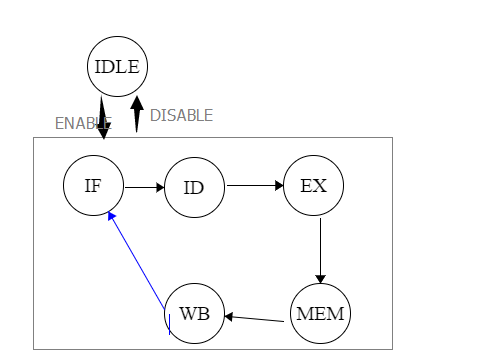
Look at the wave form, 4 clock cycles after instruction X”3C030006” FETCHED (in the WB step), the required instant value 393216 write to the register 3.

And the instruction X”00231820” add reg1 and reg3, then write back the result to reg3.

## Controller

Making the simple one cycle controller into 5 stages

State Machine Transition Diagram:



CODE

|  |
| --- |
| **architecture** Behavioral **of** MIPS\_main\_controller **is**  **type** state\_type **is** (IDLE, IFCH, ID, EX, MEM, WB);  **signal** state : state\_type := IDLE;    **begin**  decoder\_and\_output : **process**(state, Opcode) -- using asynchronous or synchronous ? the control signals for function units are mostly synchronous  **variable** PC\_en\_tmp, IF\_en\_tmp, regdst\_tmp, alusrc\_tmp, memtoreg\_tmp, regwrite\_tmp, memread\_tmp, memwrite\_tmp, branch\_tmp, jump\_tmp, rf\_writesrc\_tmp : std\_logic := '0';  **variable** aluop\_tmp : std\_logic\_vector ( 1 **downto** 0 ) := "00";  **begin**  decode : **case** Opcode **is**  -- R format  **when** "000000" =>  regdst\_tmp := '1'; alusrc\_tmp := '0'; memtoreg\_tmp := '0';  regwrite\_tmp := '1'; memread\_tmp := '0'; memwrite\_tmp := '0';  branch\_tmp := '0'; jump\_tmp := '0';  aluop\_tmp := "10"; rf\_writesrc\_tmp := '0';  -- LW  **when** "100011" =>  regdst\_tmp := '0'; alusrc\_tmp := '1'; memtoreg\_tmp := '1';  regwrite\_tmp := '1'; memread\_tmp := '1'; memwrite\_tmp := '0';  branch\_tmp := '0'; jump\_tmp := '0'; rf\_writesrc\_tmp := '0';  aluop\_tmp := "00";  -- SW  **when** "101011" =>  regdst\_tmp := '-'; alusrc\_tmp := '1'; memtoreg\_tmp := '-';  regwrite\_tmp := '0'; memread\_tmp := '0'; memwrite\_tmp := '1';  branch\_tmp := '0'; jump\_tmp := '0'; rf\_writesrc\_tmp := '0';  aluop\_tmp := "00";  -- BEQ  **when** "000100" =>  regdst\_tmp := '-'; alusrc\_tmp := '0'; memtoreg\_tmp := '-';  regwrite\_tmp := '0'; memread\_tmp := '0'; memwrite\_tmp := '0';  branch\_tmp := '1'; jump\_tmp := '0'; rf\_writesrc\_tmp := '0';  aluop\_tmp := "01";  -- JUMP  **when** "000010" =>  regdst\_tmp := '-'; alusrc\_tmp := '-'; memtoreg\_tmp := '-';  regwrite\_tmp := '0'; memread\_tmp := '0'; memwrite\_tmp := '0';  branch\_tmp := '-'; jump\_tmp := '1'; rf\_writesrc\_tmp := '0';  aluop\_tmp := "--";  -- LUI  **when** "001111" =>  regdst\_tmp := '-'; alusrc\_tmp := '-'; memtoreg\_tmp := '-';  regwrite\_tmp := '1'; memread\_tmp := '0'; memwrite\_tmp := '0';  branch\_tmp := '0'; jump\_tmp := '0'; -- PC add 1  rf\_writesrc\_tmp := '1';  aluop\_tmp := "--";  -- EXCEPTION  **when** **others** =>  regdst\_tmp := '0'; alusrc\_tmp := '0'; memtoreg\_tmp := '0';  regwrite\_tmp := '0'; memread\_tmp := '0'; memwrite\_tmp := '0';  branch\_tmp := '0'; jump\_tmp := '0'; rf\_writesrc\_tmp := '0';  aluop\_tmp := "00";  **end** **case**;    output : **case** state **is**  **when** IDLE =>  PC\_en <= '0';  IF\_en <= '0';  regdst <= '0';  alusrc <= '0';  memtoreg <= '0';  regwrite <= '0';  memread <= '0';  memwrite <= '0';  branch <= '0';  Jump <= '0';  RF\_WriteSrc <= '0';  ALUOp <= "00";    **when** IFCH =>  PC\_en <= '0';  IF\_en <= '1';  regdst <= '0';  alusrc <= '0';  memtoreg <= '0';  regwrite <= '0';  memread <= '0';  memwrite <= '0';  branch <= '0';  Jump <= '0';  RF\_WriteSrc <= '0';  ALUOp <= "00";    **when** ID =>  PC\_en <= '0';  IF\_en <= '0';  regdst <= regdst\_tmp;  alusrc <= alusrc\_tmp;  memtoreg <= memtoreg\_tmp;  regwrite <= '0';  memread <= '0';  memwrite <= '0';  branch <= branch\_tmp;  Jump <= jump\_tmp;  RF\_WriteSrc <= rf\_writesrc\_tmp;  ALUOp <= "00";    **when** EX =>  PC\_en <= '0';  IF\_en <= '0';  regdst <= regdst\_tmp;  alusrc <= alusrc\_tmp;  memtoreg <= memtoreg\_tmp;  regwrite <= '0';  memread <= '0';  memwrite <= '0';  branch <= branch\_tmp;  Jump <= jump\_tmp;  RF\_WriteSrc <= rf\_writesrc\_tmp;  ALUOp <= aluop\_tmp;    **when** MEM =>  PC\_en <= '0';  IF\_en <= '0';  regdst <= regdst\_tmp;  alusrc <= alusrc\_tmp;  memtoreg <= memtoreg\_tmp;  regwrite <= '0';  memread <= memread\_tmp;  memwrite <= memwrite\_tmp;  branch <= branch\_tmp;  Jump <= jump\_tmp;  RF\_WriteSrc <= rf\_writesrc\_tmp;  ALUOp <= aluop\_tmp;    **when** WB =>  PC\_en <= '1';  IF\_en <= '0';  regdst <= regdst\_tmp;  alusrc <= alusrc\_tmp;  memtoreg <= memtoreg\_tmp;  regwrite <= regwrite\_tmp;  memread <= memread\_tmp;  memwrite <= memwrite\_tmp;  branch <= branch\_tmp;  Jump <= jump\_tmp;  RF\_WriteSrc <= rf\_writesrc\_tmp;  ALUOp <= aluop\_tmp;    **when** **others** =>  PC\_en <= '0';  IF\_en <= '0';  regdst <= '0';  alusrc <= '0';  memtoreg <= '0';  regwrite <= '0';  memread <= '0';  memwrite <= '0';  branch <= '0';  Jump <= '0';  RF\_WriteSrc <= '0';  ALUOp <= "00";    **end** **case**;  **end** **process**;    next\_state: **process**(clk)  **begin**  **if** rising\_edge(clk) **then**  **if**(proc\_en = '0') **then** state <= IDLE;  **elsif** proc\_en = '1' **then**  **if** state = IDLE **then** state <= IFCH;  **elsif** state = IFCH **then** state <= ID;  **elsif** state = ID **then** state <= EX;  **elsif** state = EX **then** state <= MEM;  **elsif** state = MEM **then** state <= WB;  **elsif** state = WB **then** state <= IFCH; -- don't forget  **end** **if**;  **end** **if**;  **end** **if**;  **end** **process**;    **end** Behavioral; |

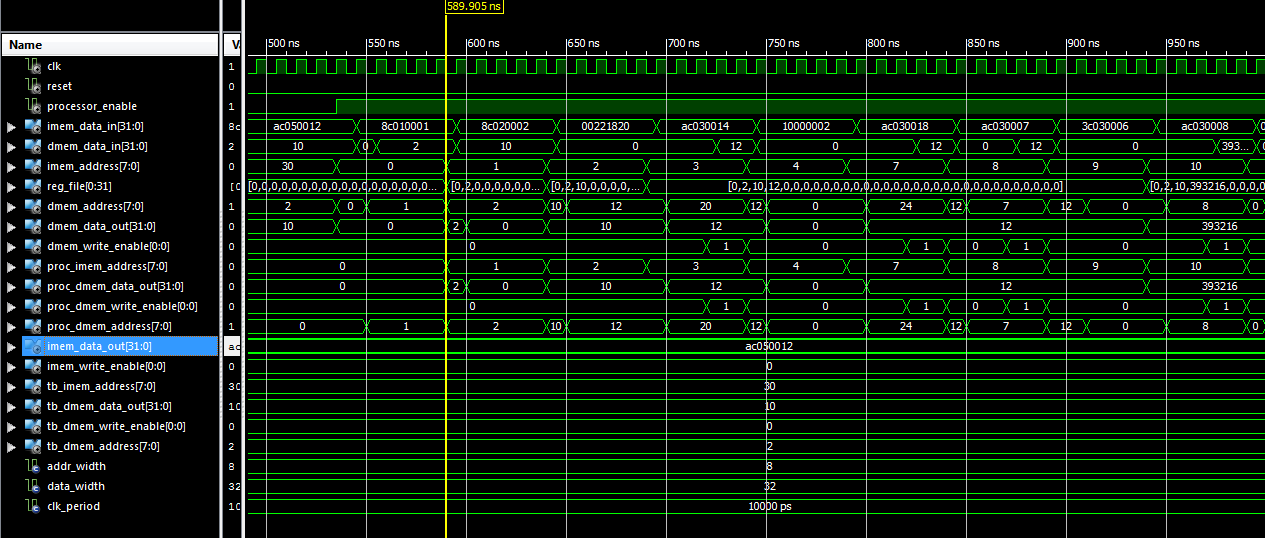
The basic Ideas is separate the whole part into two parts: next\_state sequential part and the output locgic part.

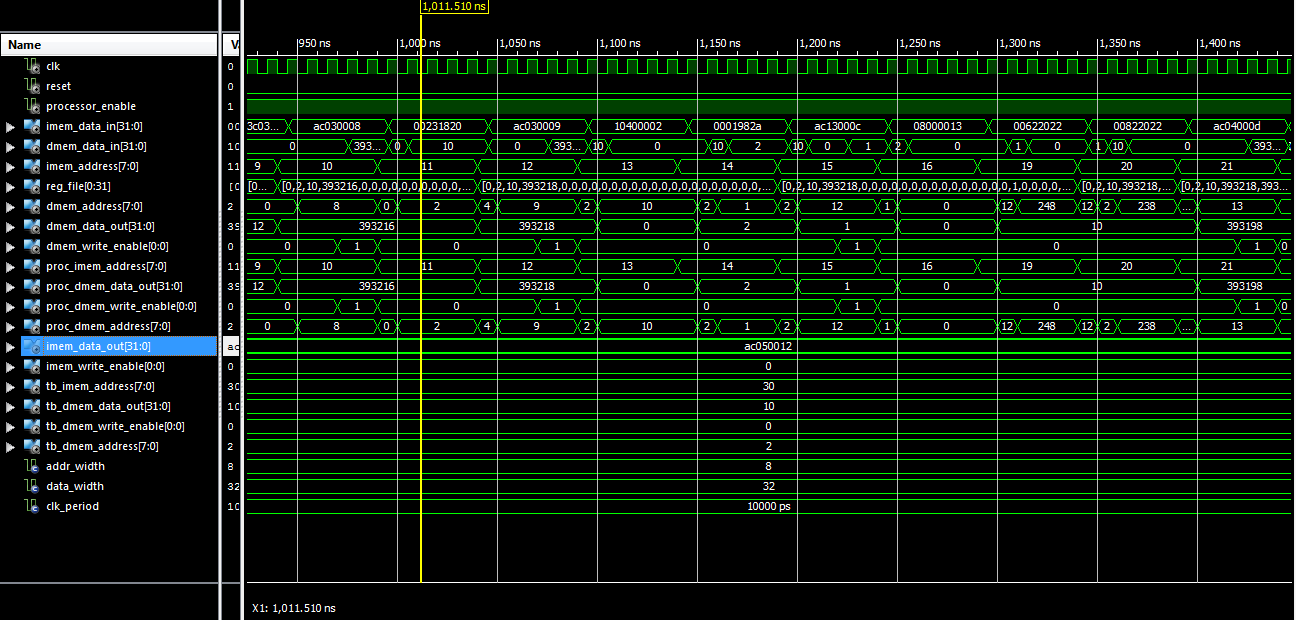
While in the ouput logic part, I separate the pure combinational input ouput logic in two main sequential steps:

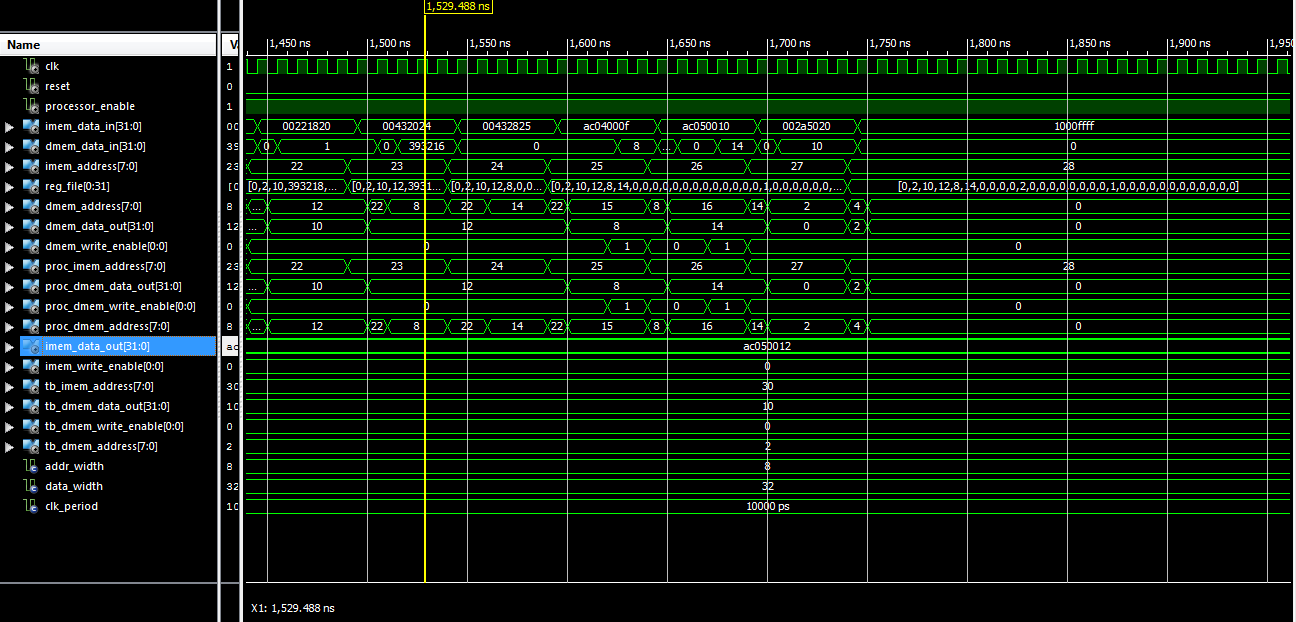
Decode and then Assign the Output port. This is intended to make the code more readable and maintainable, though may generate several more internal logic gates in the Sythesis process. But the benefit is obvious, I can add in new instruction decoding logic in the code without too much modification.

# System Test

The Testbench simulation wave form

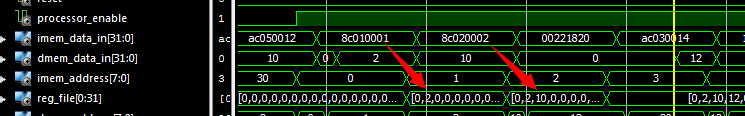






##### **LW verification**

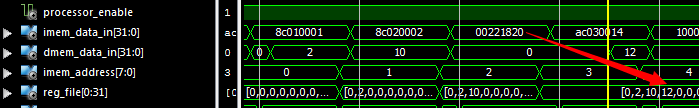




As expected.

**Add verification**

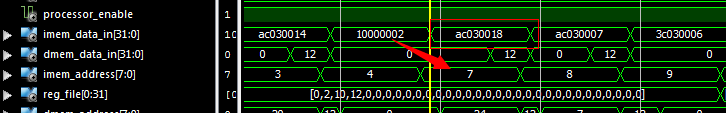




Reg3 = reg1 + reg2.

**Beq verification**

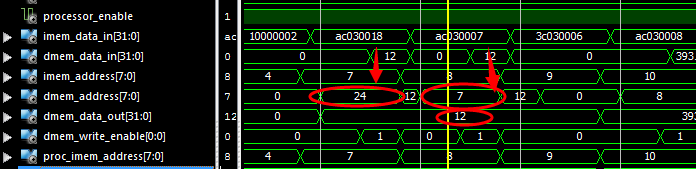




**Pc advanced from 4 to 7, skipped two instructions and fetched the one at address 7.**

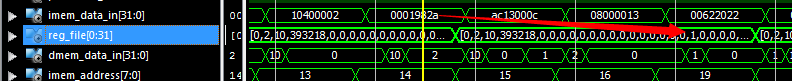
**SW verification**





**Slt verification**

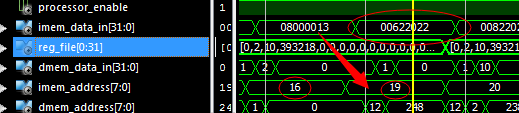




**Reg19 = 1;**

**Jump verification**

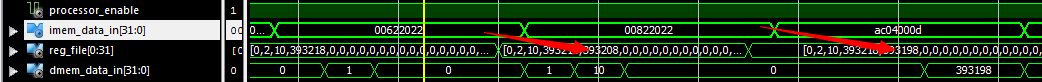




**PC jumps from 16 to 19, fetching the correct instruction**

**SUB verification**





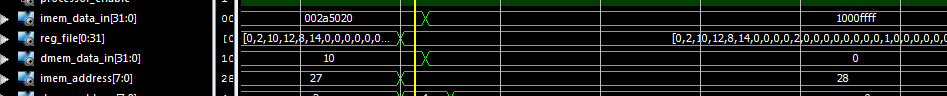
**AND / OR**





**BEQ BACKWARDS**





**PC STAYS AT 28**