EECS-361 COMPUTER ARCHITECTURE- I
GROUP PROJECT1 - SINGLE CYCLE PROCESSOR

GROUP5- MEMBERS

MATTHEW LENK
YUE WANG
NITHIN KRISHNAN

CONTENTS:

Introduction:

The single cycle processor has the ability to execute one instruction in a single clock cycle. The various parts of an instruction like, instruction fetch, decode etc. are all performed during this single cycle of the clock. Hence, the duration of clock cycle required is equal to the time taken by the largest instruction of the instruction set (In our case it is the LW instruction).

Design:

The single cycle processor is realised with the help of ModelSim software, with the test programs loaded into the memory unit of the processor. It consists mainly of an Instruction Fetch Unit, a Register file, an Arithmetic and Logic Unit and a Memory Unit.

The instruction set for this processor contains the following instructions:

1. • arithmetic: add, addi, addu, sub, subu
2. • logical: and, or, sll
3. • data transfer: lw, sw
4. • conditional branch: beq, bne, bgtz, slt, sltu.

The different types of instructions in the MIPS architecture is

1. R-type instruction
2. I-type instruction
3. J-type instruction

The instruction fetch unit gets the new instruction from the Program Counter and decodes it.

The register file stores the input operands for the operation to be performed as well any data that has to be written back.

The ALU performs the required arithmetic or logic operation ,which is selected by an ALU control, and produces an output.

Finally, data is stored in the memory unit, which can then be written back to the register file depending on the instruction.

Main Entities:

Instruction Fetch: The instruction fetch unit is tasked with taking the instruction from memory and decoding the instruction into opcode, CPU control signals, type of instruction etc.

Register File: The register file stores the operands in its memory and forwards the data to the ALU for processing. The register file also has a 32 bit write back register for data to be written back to it from the memory depending on the instruction.

Arithmetic and Logic Unit: The arithmetic and logic unit takes in 2 operands and performs the required operation on them as specified by the opcode. The opcode generates the ALU control signals,which in turn control the operation performed by the ALU.

VHDL Implementation

```
begin

--For choosing which overflow and carryOut signal to use
--The only two options for which operation it would come from
--is addition or subtraction.
--I check if its addition whose opcode is "010"
--Then I use that result as a selector for the muxes
--to choose whether to output the addOverflow and carryout or the subtraction ones
ctrl1: not_gate port map(ctrl(0), o1);
ctrl2: not_gate port map(ctrl(2), o2);
ctrl3: and_gate port map(o1, o2, o3);
ctrl4: and_gate port map(o3, ctrl(1), addOrSub);



result<= tempResult;

--Multiplexers to choose which operation should be outputed.
muxResult: mux_4 generic map(n => 32)
                port map(
                        sel    => ctrl,
                        addOp  => resultAdd,
                        subOp  => resultSub,
                        andOp  => resultAnd,
                        orOp   => resultOr,
                        sllOp  => resultSll,
                        sltOp  => resultSlt,
                        sltuOp => resultSltu,
                        Result => tempResult
                        );
```

```
muxCarryOut: mux
                port map(
                        sel    => addOrSub,
                        src0  => carryOutSub,
                        src1  => carryOutAdd,
                        z => carryOut
                        );

muxOverflow: mux
                port map(
                        sel   => addOrSub,
                        src0  => overflowSub,
                        src1  => overflowAdd,
                        z => overFlow
                        );

--calculating the zero value from the result that will be outputed of the ALU
zeroPort: zeroDetector port map(tempResult, zero);



--Each operation has its own entity

andOperation: and_gate_32 port map(A, B, resultAnd);
orOperation: or_gate_32 port map(A,B, resultOr);



addEntity: adder32 port map(A=>A, B=>B, carryIn=> '0', carryOut=>carryOutAdd,
overFlow=>overflowAdd,  result=>resultAdd);
subEntity: sub32 port map(A=>A, B=>B, carryOut=>carryOutSub,
overFlow=>overflowSub,  result=>resultSub);
resultSlt<= (0=> resultSub(31), others=>'0');

notCarry: not_gate port map(carryOutSub, notCarryOutSub);
resultSltu<= (0=>notCarryOutSub, others=>'0');

sllEntity: SLLmod port map (A=>B, B=>A,  result=>resultSll);
```

```
end architecture structural;
```

Memory: The data from the ALU is written into the data memory. This result is stored there, or written back to the register file, depending on the instruction type.
VHDL implementation:

```
                begin
--clk_inv : not_gate port map(x => clk, z => clk_n);
sram_map: sram
generic map(mem_file => data_mem_file)
port map(
                 cs => '1',
                 oe => '1',
                 we => we,
                 addr => addr,
                 din => data_in,
                 dout => temp_data_out);

data_out<=temp_data_out;

end structural;
```

Top-Level Entity: The top level entity(datapath), connects all the components together.
VHDL IMplementation:

```
                begin

opcode<= inst(31 downto 26);--from IFU
Rt<=inst(20 downto 16);
Rs<=inst(25 downto 21);
Rd<=inst(15 downto 11);
Ra<=Rs;
Rb<=Rt;
immed<=inst(15 downto 0);
func<=inst(5 downto 0);
shamt(4 downto 0)<=inst(10 downto 6);
shamt(31 downto 5)<= (others=>'0');
```

```
--Read_data1<=opA;
opA<= Read_data1;

I1: instructMem generic map( mem_file=>mem_file) port map(address,inst);
I2: main_control port
map(opcode,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,Branch,Jump,ExtOp,ALUop);-
-Getting main control signals from opcode
I3: mux_n generic map(n=> 5 ) port map(RegDst,Rt,Rd,Rw);-- selecting input for
register
I4: register_file port
map(clk,reset,RegWr,Ra,Rb,Rw,Write_data,temp_read,Read_data2);-- output from
register file

I5: signExtender port map(immed,signExtout);-- calculating extended output
I6: ALU_control port map(opcode, func,ALUop,ALUctr);-- calculating alu control signals
NOR0: nor_gate port map(ALUctr(1),ALUctr(0),AC);
AND1: and_gate port map(ALUctr(2),AC,Ensll);
MUX1: mux_32 port map(Ensll,temp_read,shamt,Read_data1);--temp_read is read
from data register and MUXd with shamt to produce read_data1 which is given as opA
to ALU.
I7: mux_32 port map(ALUSrc,Read_data2,signExtout,opB);-- selecting operand B for
ALU operations
I8: ALU port map(opA,opB,ALUctr,carryout,overflow,zero_f,result);-- alu values
I9: not_gate port map(MemWr,MemRead);--generating memory read signal for the
memory file
cs<='1';
I10: data_memory generic map(data_mem_file=>mem_file) port
map(Read_data2,result,MemWr,Dout);--Writing data to memory
I11: mux_32 port map(MemtoReg,result,Dout,Write_data);-- selecting which data to
write back
I12: adder32 port
map(A=>address,B=>"00000000000000000000000000000100",carryIn=>'0',result=>P
Cplus4);
--I13: getJumpAddress port map(inst(25 downto 0),PCplus4(31 downto
28),JumpAddress);
--I14: mux_32 port map(Jump,PCplus4,JumpAddress,nextAddress);
I13: and_gate port map(branch, zero_f, branchCtrl );
I14: adder32 port map(A=> PCplus4, B(31 downto 2) => signExtout(29 downto 0), B(1
downto 0) =>"00", carryIn=>'0', result=> branchAddress );
```

```
I16: mux_32 port map(branchCtrl, PCplus4, branchAddress, nextAddress);
I15: PC port map(nextAddress,address,clk);

-- Call PC+4 or New address based on jump.

end architecture structural;
```

Main Control Signals: The main control signals are generated with the help of the OPcode. These signals help select which data is to be provided as input and if data should be written back or read from.
The various control signals are:

I. RegDst: Selects either a 5 bit operand or 16 bit immediate value depending on the type of instruction.
II. ALUSrc: Selects 32 bit data from the register to perform ALU operations or selects 32 bit input to perform shift instructions.
III. MemtoReg: Selects either the ALU output or data stored in the data memory depending on the instruction.
IV. RegWr: If set, this signal enables writing of data on to the registers.
V. MemWr: If set, this signals enable writing onto the data memory unit.
VI. Branch: Helps select the address for the next instruction. If branch is taken, a new address is calculated. Otherwise, the next instruction is loaded to the Program Counter
VII. ALUop: Controls the operation of the ALU.

VHDL Implementation:

```
--Find Instruction Type based on Opcode

--Rtype
r1: six_in_and_gate port map(nop5,nop4,nop3,nop2,nop1,nop0,inst(0));

--SLL

--OR Immediate
or1: six_in_and_gate port
map(nop5,nop4,opcode(3),opcode(2),nop1,opcode(0),inst(1));
--Load word
addi: six_in_and_gate port map(nop5, nop4, opcode(3), nop2, nop1, nop0, inst(8) );

lw1: six_in_and_gate port
map(opcode(5),nop4,nop3,nop2,opcode(1),opcode(0),inst(2));
```

```
--Store Word

sw1: six_in_and_gate port
map(opcode(5),nop4,opcode(3),nop2,opcode(1),opcode(0),inst(3));

-- Breanch Equal

beq1: six_in_and_gate port map(nop5,nop4,nop3,opcode(2),nop1,nop0,inst(4));
bne1: six_in_and_gate port map(nop5,nop4,nop3,opcode(2),nop1,opcode(0),inst(7));
bgtz1: six_in_and_gate port
map(nop5,nop4,nop3,opcode(2),opcode(1),opcode(0),inst(6));

-- Jump

jmp1: six_in_and_gate port map(nop5,nop4,nop3,nop2,opcode(1),nop0,inst(5));

--RegWrite

regwr1: or_gate port map(inst(0),inst(1),temp0);
regwr2: or_gate port map(inst(2),temp0,temp3);
regwr3: or_gate port map(temp3, inst(8), RegWr);


--ALUSRC

alusrc1: or_gate port map(inst(1),inst(2),temp1);
alusrc2: or_gate port map(temp1,inst(3),temp2);
alusrc3: or_gate port map(temp2, inst(8), ALUSrc);
--RegDst

RegDst<=inst(0);

--MemtoReg

MemtoReg<=inst(2);

--MemWrite
```

```
MemWr<=inst(3);

--Branch
----BEQ
--Branch(2)<=inst(4);
----BNE
--Branch(1)<=inst(7);
----BGTZ
--Branch(0)<=inst(6);

branchCheck1: or_gate port map (inst(4), inst(7), b1);
branchCheck2: or_gate port map(b1, inst(6), Branch);

--Jump
Jump<=inst(5);
--ExtOP

extop1: or_gate port map(inst(2),inst(3),ExtOp);

--ALUop

ALUop(2)<=inst(0);
ALUop(1)<=inst(1);
ALUop(0)<=inst(4);
```

Main Control Signal Values:

OPCODE	000000	001101	100011	101011	000100	000101	000110	000010
	R-TYPE	ORI	LW	SW	BEQ	BNE	BGTZ	JMP
RegDst	1	0	0	X	X	X	X	X
ALUSrc	0	1	1	1	0	0	0	X
MemtoReg	0	0	1	X	X	X	X	X
MemWr	1	1	1	0	0	0	0	0
RegWr	0	0	0	1	0	0	0	0
Branch	0	0	0	0	1	1	1	0

Jump	0	0	0	0	0	0	0	1
ExtOP	X	0	1	1	X	X	X	X
ALUop	R-type	Or	Add	Add	Subt	Subt	Subt	XXX

ALU Control Signals: The ALU control signals or ALUop controls the operation of the ALu. It helps select which operation needs to be performed on the two input operands.

VHDL Implementation:

```
--finding bit 2 of ALUctr

a21: three_in_and_gate port
map(inv_ALUop(2),inv_ALUop(1),ALUop(0),temp_signals(0));
a22: four_in_and_gate port
map(ALUop(2),inv_func(2),func(1),inv_func(0),temp_signals(1));
a23: or_gate port map(temp_signals(0),temp_signals(1),alsotemp(0));
a25: five_in_and_gate port
map(ALUop(2),func(3),inv_func(2),inv_func(1),func(0),alsotemp(1));
a26: or_gate port map(alsotemp(0),alsotemp(1),temp_ALUctr(2));

-- Finding bit1 of ALUctr

a11: and_gate port map(inv_ALUop(2),inv_ALUop(1),temp_signals(2));
a12: three_in_and_gate port map(ALUop(2),inv_func(2),inv_func(0),temp_signals(3));
a13: or_gate port map(temp_signals(2),temp_signals(3),alsotemp(2));
a14: five_in_and_gate port
map(ALUop(2),inv_func(3),func(2),func(1),inv_func(0),alsotemp(3));
a15: or_gate port map(alsotemp(2),alsotemp(3),temp_ALUctr(1));


-- Finding bit0 of ALUctr

a01: and_gate port map(inv_ALUop(2),ALUop(1),temp_signals(4));
a02: five_in_and_gate port
map(ALUop(2),inv_func(3),func(2),inv_func(1),func(0),temp_signals(5));
```

a03: five_in_and_gate port
map(ALUop(2),func(3),inv_func(2),func(1),inv_func(0),temp_signals(6));
a04: or_gate port map(temp_signals(5),temp_signals(4),temp_signals(7));
a05: or_gate port map(temp_signals(6),temp_signals(7),alsotemp(4));
a06: five_in_and_gate port
map(ALUop(2),inv_func(3),func(2),func(1),inv_func(0),alsotemp(5));
a07: or_gate port map(alsotemp(4),alsotemp(5),temp_ALUctr(0));

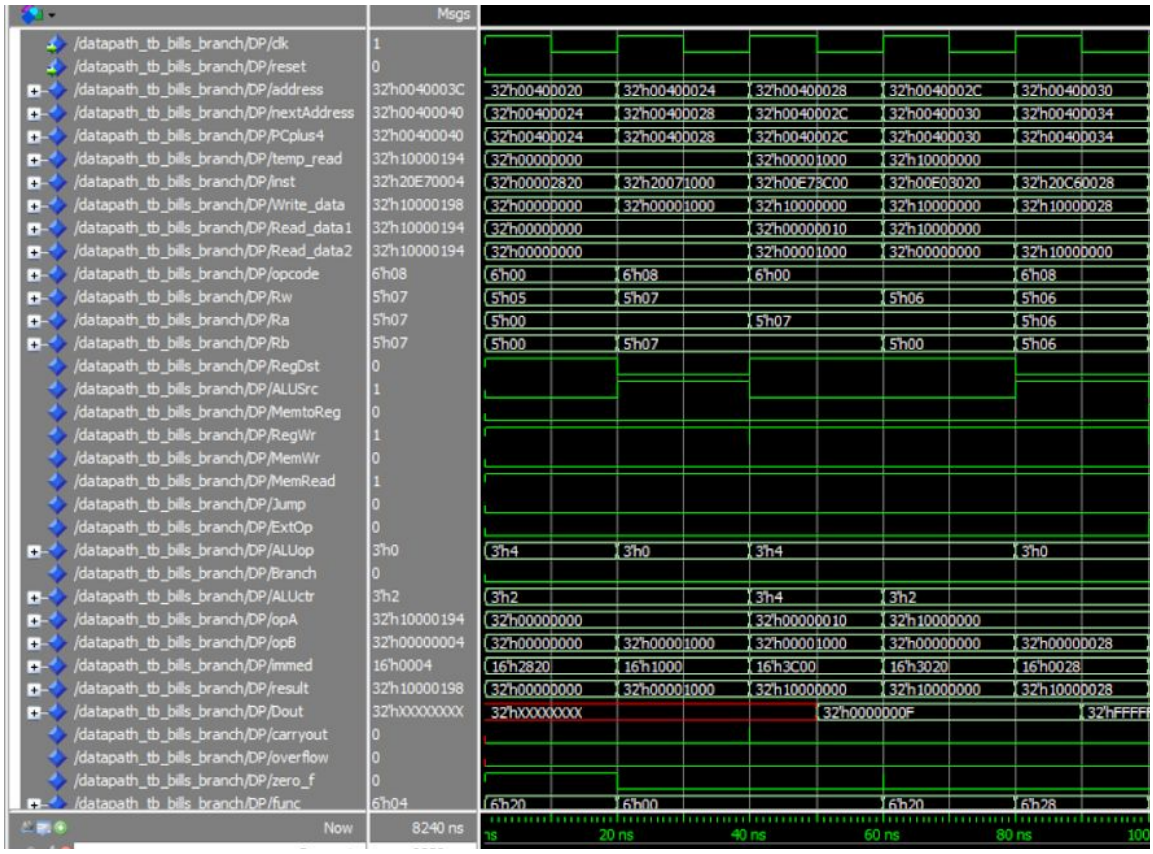
The ALUop is a three bit value. So, we can perform up to 7 distinct operations.

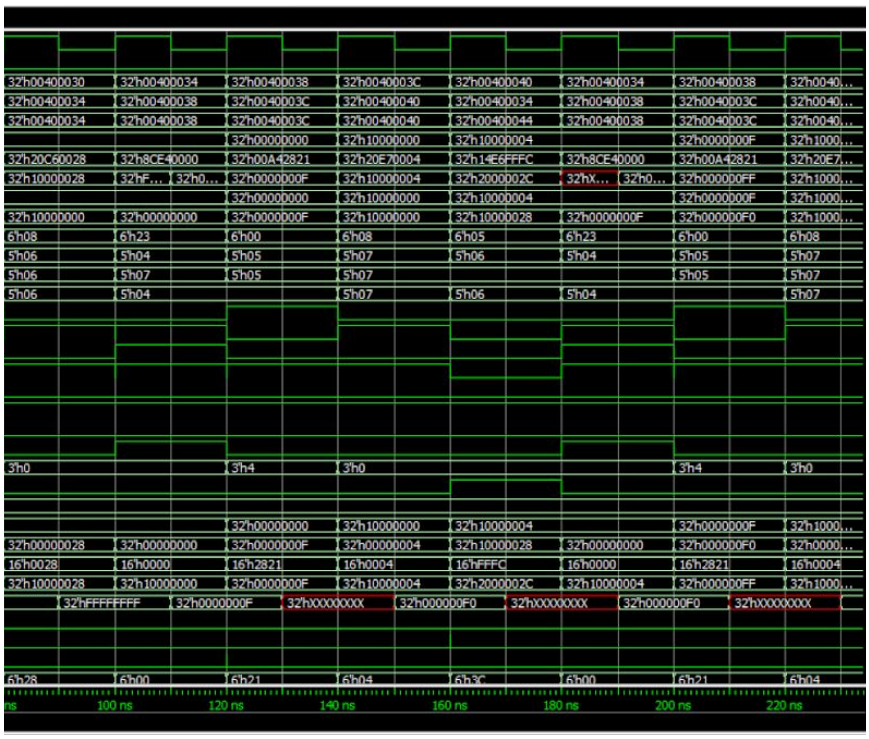
ALU Control Signal Values:

ALUop			Func				ALUctr			Operation
0	0	0	X	X	X	X	0	1	0	ADD
0	0	1	X	X	X	X	1	1	0	SUB
0	1	0	X	X	X	X	0	0	1	OR
1	X	X	0	0	0	0	0	1	0	ADD
1	X	X	0	0	1	0	1	1	0	SUB
1	X	X	0	1	0	0	0	0	0	AND
1	X	X	0	1	0	1	0	0	1	OR
1	X	X	1	0	1	0	1	1	1	SLT
1	X	X	1	0	0	0	0	1	0	ADDU
1	X	X	1	1	1	0	1	1	0	SUBU
1	X	X	1	0	0	1	1	0	0	SLL
1	X	X	1	0	1	1	0	1	1	SLTU

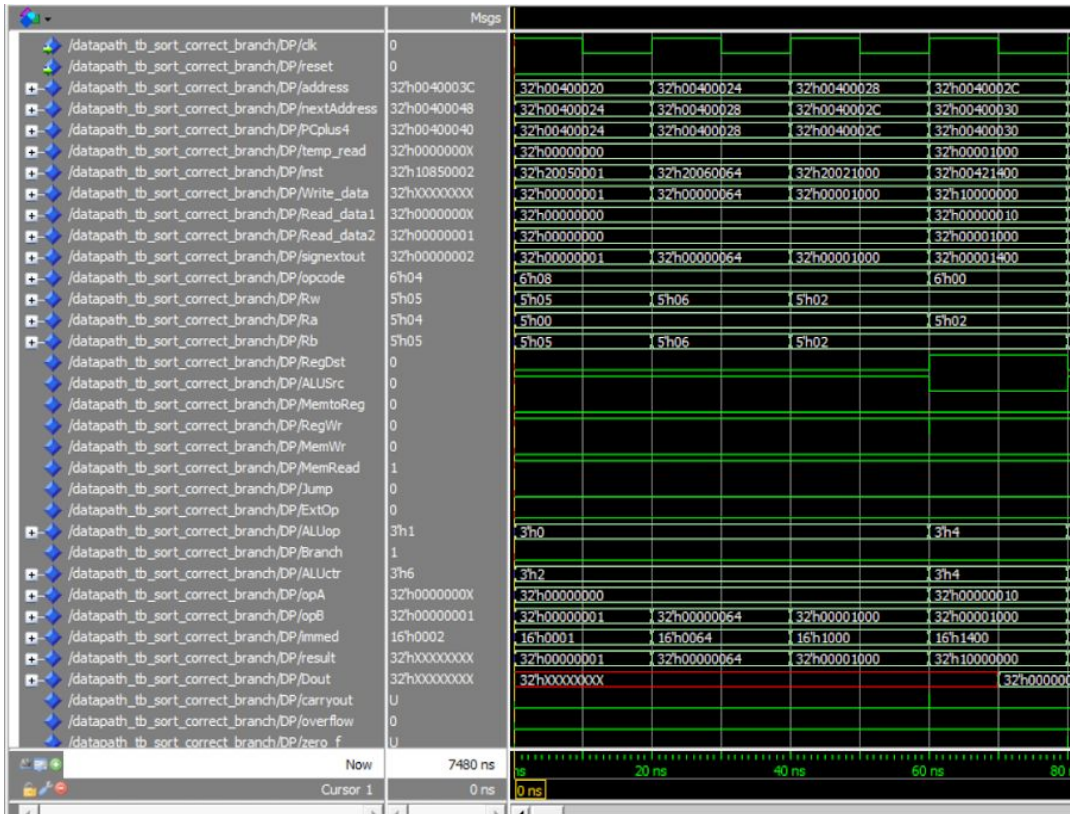
Simulation:

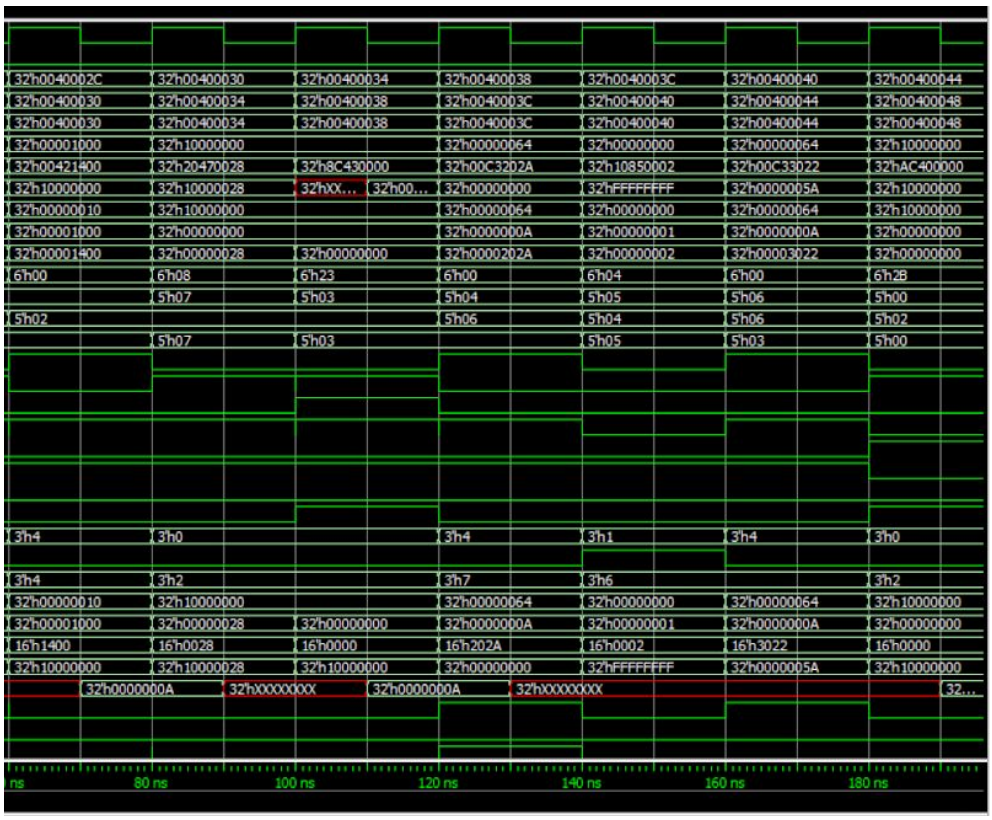
Unsigned Sum



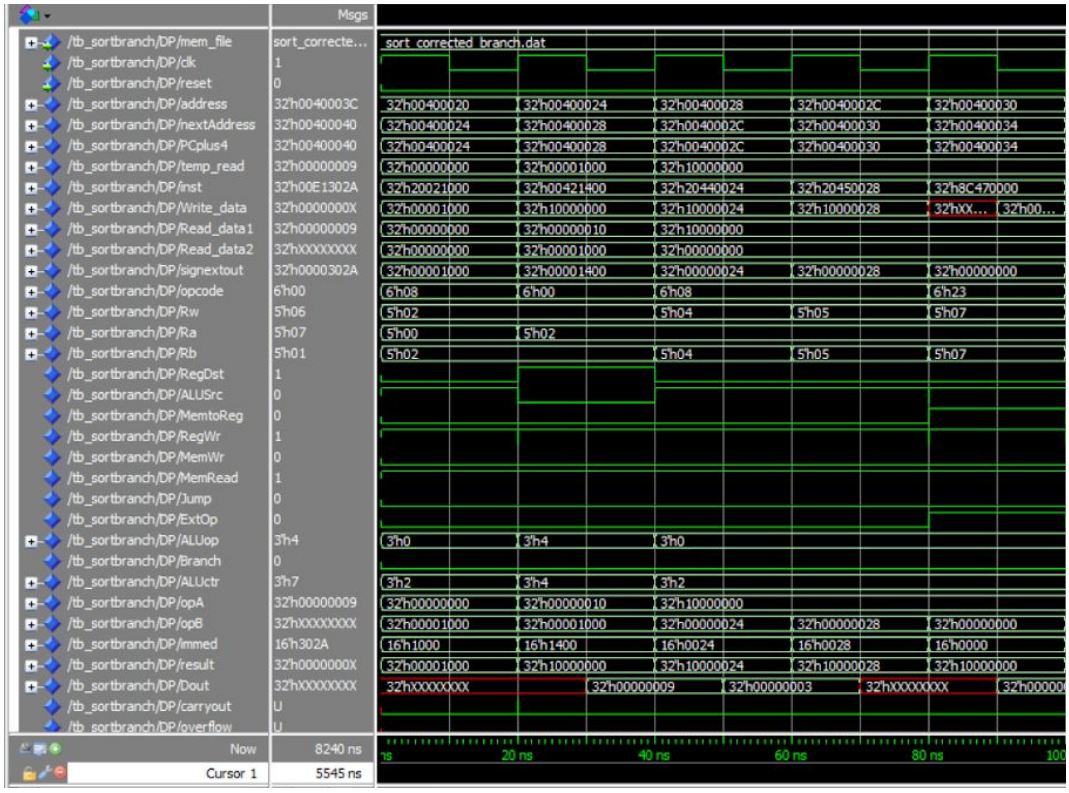


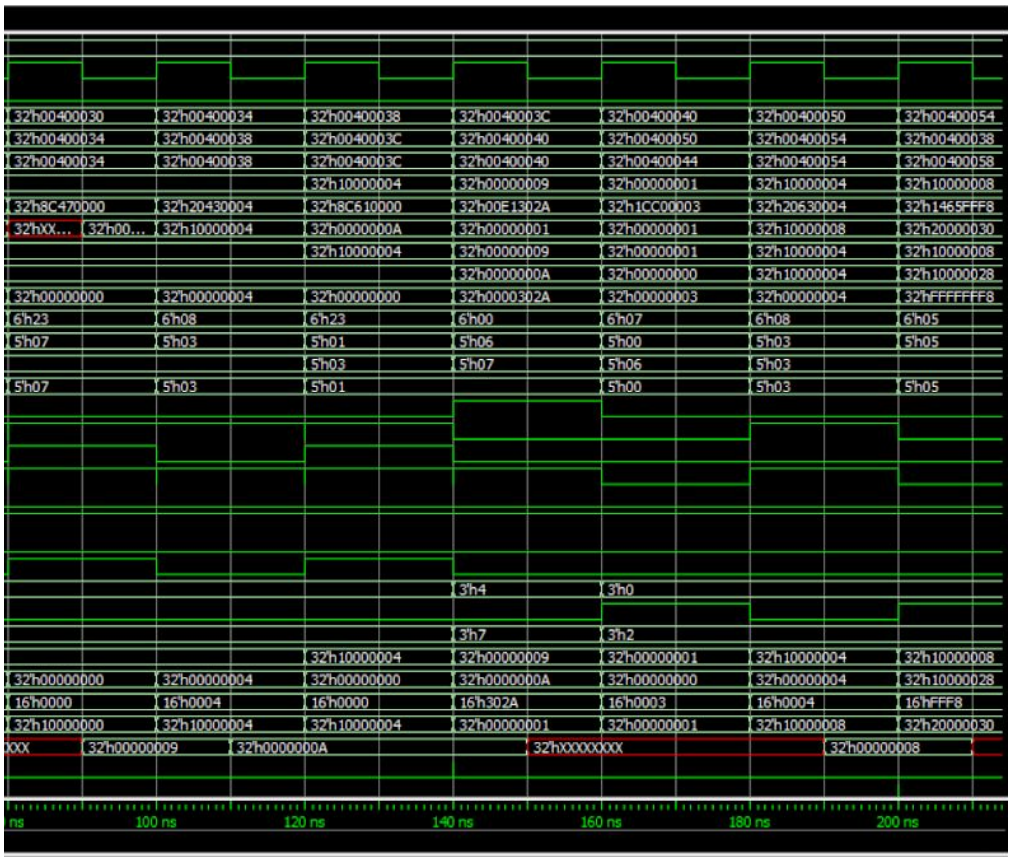
Bills Branch





Sort Branch


Msgs
/tb_sortbranch/DP/mem_file
sort_corrected_branch.dat
/tb_sortbranch/DP/clk
/tb_sortbranch/DP/reset
/tb_sortbranch/DP/address
/tb_sortbranch/DP/nextAddress
/tb_sortbranch/DP/PCplus4
/tb_sortbranch/DP/temp_read
/tb_sortbranch/DP/inst
/tb_sortbranch/DP/Write_data
/tb_sortbranch/DP/Read_data1
/tb_sortbranch/DP/Read_data2
/tb_sortbranch/DP/signextout
/tb_sortbranch/DP/opcode
/tb_sortbranch/DP/Rw
/tb_sortbranch/DP/Ra
/tb_sortbranch/DP/Rb
/tb_sortbranch/DP/RegDst
/tb_sortbranch/DP/ALUSrc
/tb_sortbranch/DP/MemtoReg
/tb_sortbranch/DP/RegWr
/tb_sortbranch/DP/MemWr
/tb_sortbranch/DP/MemRead
/tb_sortbranch/DP/Jump
/tb_sortbranch/DP/ExtOp
/tb_sortbranch/DP/ALUop
/tb_sortbranch/DP/Branch
/tb_sortbranch/DP/ALUctr
/tb_sortbranch/DP/opA
/tb_sortbranch/DP/opB
/tb_sortbranch/DP/immed
/tb_sortbranch/DP/result
/tb_sortbranch/DP/Dout
/tb_sortbranch/DP/carryout
/tb_sortbranch/DP/overflow
Now
8240 ns
Cursor 1
5545 ns
20 ns
40 ns
60 ns
80 ns


100 ns
120 ns
140 ns
160 ns
180 ns
200 ns

Results:

Our three programs all appeared to be functioning all though the became stuck in an infinite loop which we were not able to figure out in time. We received the correct solutions albeit the programs would not exit at the appropriate time.