**Lab 4 Report**

First Uber Cool Design ISA

By: Brandon Chin (A11845998), Leandro Lubrico (A12077909), and Kevin Chen (A10623152)

**Top Verilog Module**

module top (

input start,

input [6:0] start\_address,

input clock,

output logic done

);

// START WIRES

logic [6:0] next;

logic [6:0] current;

logic [8:0] instruction;

logic [7:0] regData1;

logic [7:0] regData2;

logic [7:0] aluData;

logic overflow;

logic [7:0] memData;

logic [7:0] writeData;

logic branch;

logic taken;

logic halt;

logic regWrite;

logic regDest;

logic memRead;

logic memWrite;

logic memToReg;

// END WIRES

// START MODULES

PC program\_counter (

clock,

next,

current

);

fetch fetch\_unit (

start,

start\_address,

branch,

taken,

instruction[4:0],

halt,

current,

next

);

instruction\_ROM instr\_mem (

current,

instruction

);

reg\_file register\_file (

start,

start\_address,

clock,

regWrite,

regDest,

instruction[5:3],

instruction[2:0],

writeData,

regData1,

regData2

);

ALU alu (

clock,

instruction[8:6],

instruction[5:3],

instruction[5],

regData1,

regData2,

aluData,

overflow

);

data\_memory data\_mem (

start,

clock,

memRead,

memWrite,

regData2,

regData1,

memData

);

WriteDataSwitch data\_switch (

memToReg,

aluData,

memData,

writeData

);

Control control (

instruction[8:6],

instruction[5:3],

instruction[5],

overflow,

branch,

taken,

halt,

regWrite,

regDest,

memRead,

memWrite,

memToReg

);

// END MODULES

always\_comb begin

done = halt;

end

endmodule

**PC Verilog Module**

module PC (

input clock,

input [6:0] next,

output logic [6:0] current

);

// update program counter every clock cycle

always\_ff @(posedge clock) begin

current = next;

end

endmodule

**Fetch Verilog Module**

module fetch (

input start,

input [6:0] start\_address,

input branch,

input taken,

input [4:0] offset, // target

input halt,

input [6:0] current,

output logic [6:0] next

);

logic signed [7:0] signed\_address; // for branch calculations

logic signed [4:0] signed\_offset;

always\_comb begin

signed\_address = current;

signed\_offset = offset;

// START NEXT INTRUCTION LOGIC

// initialize

if (start == 1) begin

next = start\_address;

end

// branch

else if ((branch == 1) && (taken == 1)) begin

next = signed\_address + signed\_offset;

end

// halt

else if (halt == 1) begin

next = current;

end

// increment

else begin

next = current + 1;

end

// END NEXT INSTRUCTION LOGIC

end

endmodule

**Instruction ROM Verilog Modules**

module instruction\_ROM #(parameter numInstr = 128) (

input [6:0] address,

output [8:0] instruction

);

logic [8:0] internal\_memory [0: numInstr-1];

initial begin

$readmemb("instructions.bin", internal\_memory);

end

assign instruction = address < numInstr ? internal\_memory[address] : 9'b0;

endmodule

**Register File Verilog Module**

module reg\_file (

input start,

input [6:0] start\_address,

input clock,

input write,

input dest,

input [2:0] src1,

input [2:0] src2,

input [7:0] writeData,

output logic [7:0] readData1,

output logic [7:0] readData2

);

logic [7:0] registers [0:7];

// read

always\_comb begin

readData1 = registers[src1];

readData2 = registers[src2];

end

always\_ff @(negedge clock) begin

// initialize

if (start) begin

case (start\_address)

// product

7'b0 : begin

$readmemb("product.rf", registers);

end

// string match

7'b0100010 : begin

$readmemb("string\_match.rf", registers);

end

// closest pair

7'b0110010 : begin

$readmemb("closest\_pair.rf", registers);

end

default: begin

registers = '{8'b0, 8'b0, 8'b0, 8'b0, 8'b0, 8'b0, 8'b0, 8'b0};

end

endcase

end

// write

if (write == 1) begin

case (dest)

0: begin

registers[src1] = writeData;

end

default: begin

registers[src2] = writeData;

end

endcase

end

end

endmodule

**ALU Verilog Module**

module ALU (

input clock,

input [2:0] opcode,

input [2:0] funcA,

input funcB,

input [7:0] in1,

input [7:0] in2,

output logic [7:0] result,

output logic overflow

);

logic [8:0] temp;

logic overflow\_reg;

// START ALIASES

// opcode

parameter

ADD = 3'b010,

MATCH = 3'b011,

LT = 3'b100,

DIST = 3'b101,

FUNCA = 3'b110,

FUNCB = 3'b111;

// funcA

parameter

LSL = 3'b000,

LSR = 3'b001,

INCR = 3'b010,

AND1 = 3'b011,

EQZ = 3'b100,

ZERO = 3'b101,

TBD = 3'b110,

HALT = 3'b111;

// funcB

parameter

BNO = 0,

BOF = 1;

// END ALIASES

always @ (\*) begin

overflow = overflow\_reg;

// add

if (opcode == ADD) begin

temp = in1 + in2;

result = temp[7:0];

end

// match

else if (opcode == MATCH) begin

result = 8'bz;

temp[8] = (in1[3:0] == in2[3:0]);

end

// less than

else if (opcode == LT) begin

result = 8'bz;

temp[8] = in1 < in2;

end

// distance

else if (opcode == DIST) begin

temp = ($signed(in1) - $signed(in2));

result = temp[8] ? -temp : temp;

end

else if (opcode == FUNCA) begin

if (funcA == LSL) begin

temp = in2;

temp = temp << 1;

result = temp[7:0];

end

else if (funcA == LSR) begin

temp = in2;

temp = temp >> 1;

result = temp[7:0];

end

else if (funcA == INCR) begin

temp = in2;

temp = temp + 1;

result = temp[7:0];

end

else if (funcA == AND1) begin

temp[8] = in2[0] & 1'b1;

result = 8'bz;

end

else if (funcA == EQZ) begin

temp[8] = (in2 == 0);

result = 8'bz;

end

else if (funcA == ZERO) begin

temp = 0;

result = temp[7:0];

end

else begin

temp[8] = 1'bz;

result = 8'bz;

end

end

else if (FUNCB) begin

temp[8] = overflow\_reg;

result = 8'bz;

end

else begin

temp[8] = 1'bz;

result = 8'bz;

end

end

// write to overflow register

always\_ff @(negedge clock) begin

if (temp[8] == 1) begin

overflow\_reg = 1;

end

else begin

overflow\_reg = 0;

end

end

endmodule

**Data Memory Verilog Module**

module data\_memory(

input start,

input clock,

input read,

input write,

input [7:0] address,

input [7:0] dataIn,

output logic [7:0] dataOut

);

logic [7:0] mem [0:255];

always\_comb begin

if (read) begin

dataOut = mem[address];

end

else begin

dataOut = 8'bz;

end

end

always\_ff @(negedge clock) begin

if (start) begin

$readmemb("default.mem", mem);

end

if (write) begin

mem[address] = dataIn;

end

end

endmodule

**Write Data Switch Verilog Module**

module WriteDataSwitch (

input memToReg,

input [7:0] aluData,

input [7:0] memData,

output logic [7:0] writeData

);

always\_comb begin

if (memToReg) begin

writeData = memData;

end

else begin

writeData = aluData;

end

end

endmodule

**Control Verilog Module**

module Control(

input [2:0] opcode,

input [2:0] funcA,

input funcB,

input overflow,

output logic branch,

output logic taken,

output logic halt,

output logic regWrite,

output logic regDest,

output logic memRead,

output logic memWrite,

output logic memToReg

);

// START ALIASES

// opcode

parameter

LOAD = 3'b000,

STORE = 3'b001,

ADD = 3'b010,

MATCH = 3'b011,

LT = 3'b100,

DIST = 3'b101,

HAS\_FUNCA = 3'b110,

HAS\_FUNCB = 3'b111;

// funcA

parameter

LSL = 3'b000,

LSR = 3'b001,

INCR = 3'b010,

AND1 = 3'b011,

EQZ = 3'b100,

ZERO = 3'b101,

TBD = 3'b110,

HALT = 3'b111;

// funcB

parameter

BNO = 0,

BOF = 1;

// END ALIASES

always\_comb begin

case (opcode)

LOAD: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 0;

memRead = 1;

memWrite = 0;

memToReg = 1;

end

STORE: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 0;

regDest = 0;

memRead = 0;

memWrite = 1;

memToReg = 0;

end

ADD: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 0;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

DIST: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 0;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

HAS\_FUNCA: begin

case (funcA)

LSL: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 1;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

LSR: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 1;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

INCR: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 1;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

ZERO: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 1;

regDest = 1;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

HALT: begin

branch = 0;

taken = 0;

halt = 1;

regWrite = 0;

regDest = 0;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

default: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 0;

regDest = 0;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

endcase

end

HAS\_FUNCB: begin

branch = 1;

if ((funcB && overflow) || !(funcB || overflow)) begin

taken = 1;

end

else begin

taken = 0;

end

halt = 0;

regWrite = 0;

regDest = 0;

memRead = 0;

memWrite = 0;

memToReg = 0;

end

default: begin

branch = 0;

taken = 0;

halt = 0;

regWrite = 0;

regDest = 0;

memRead = 0;

memWrite = 0;

memToReg = 0;

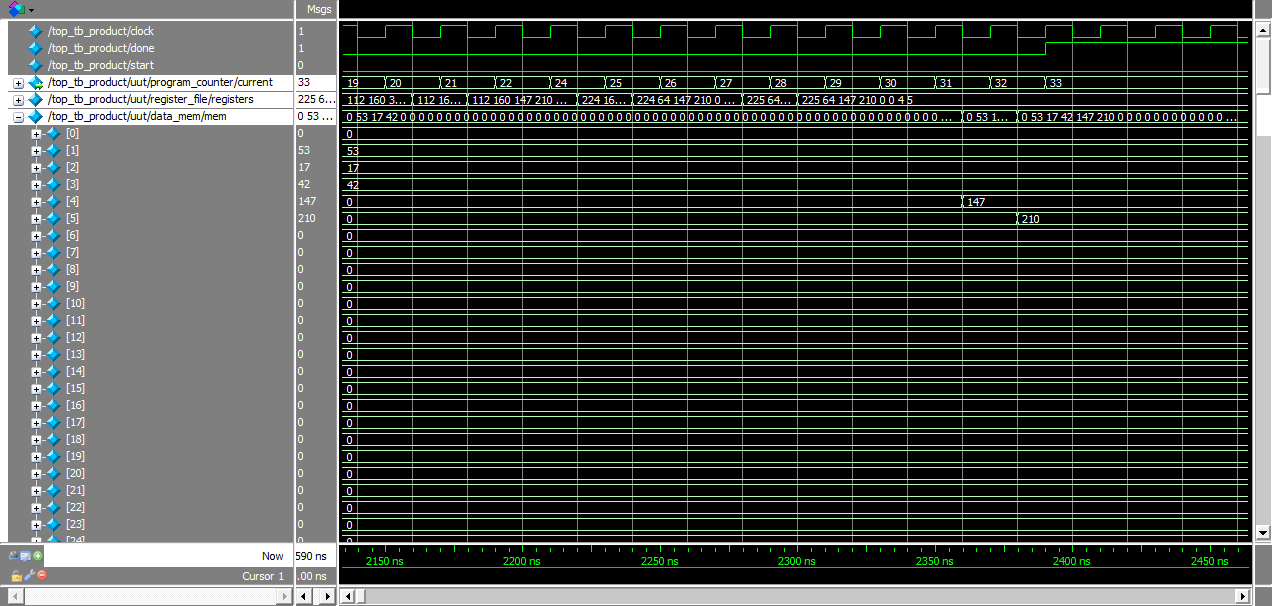
end

endcase

end

endmodule

**Product Timing Diagram**



Program Counter: The signal named “…/program\_counter/current” represents the program counter state. This just determines which instruction we are currently running from our instruction binary.

Registers The signal named “…/register\_file/registers” represents the current state of the data inside our registers. The values in the registers store the intermediate data of our product algorithm, such as when we shift the bits. The reason that the values are truncated and not expanded, like memory, is because it would only serve to clutter the timing diagram for this problem.

Memory: The signal named “…/data\_mem/mem” represents the values in memory. The initial values get stored into data locations 1 (A), 2 (B), and 3 (C). The resulting product is stored in memory 4 and 5. The higher bits of the result are stored in memory location 4 and the lower bits in memory location 5.

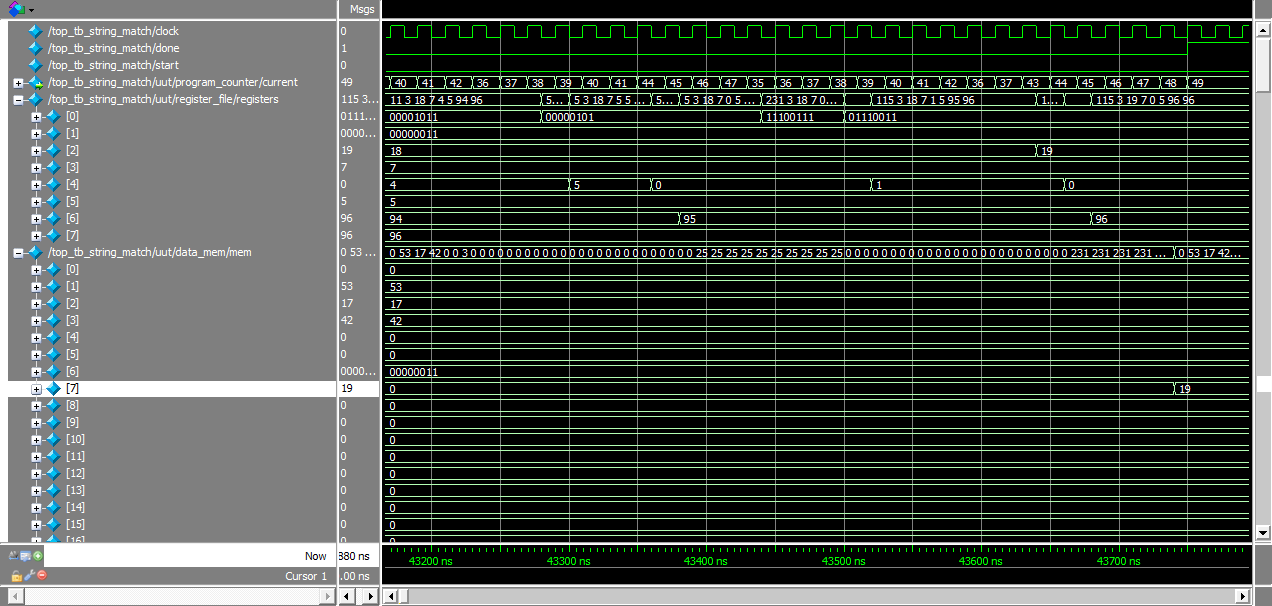
The numbers being multiplied are 53, 17, and 42.

53 \* 17 \* 42 = 37,842 = 0b1001001111010010

Mem[4] = 147 = 0b10010011 Mem[5] = 210 = 0b11010010

The correct result is achieved since concatenating the binary from Mem[4] and Mem[5] results in the binary value of 37,842

**String Match Timing Diagram**



Registers The signal named “…/register\_file/registers” represents the current state of the data inside our registers.

|  |  |
| --- | --- |
| Register | What register holds |
| 0 | The current string entry being compared to the string pattern |
| 1 | String pattern |
| 2 | Match count |
| 3 | Address to store the result |
| 4 | Inner loop index (for our algorithm) |
| 5 | Inner loop bound (for our algorithm) |
| 6 | Address of array element |
| 7 | Address of first byte in array + array length (size) |

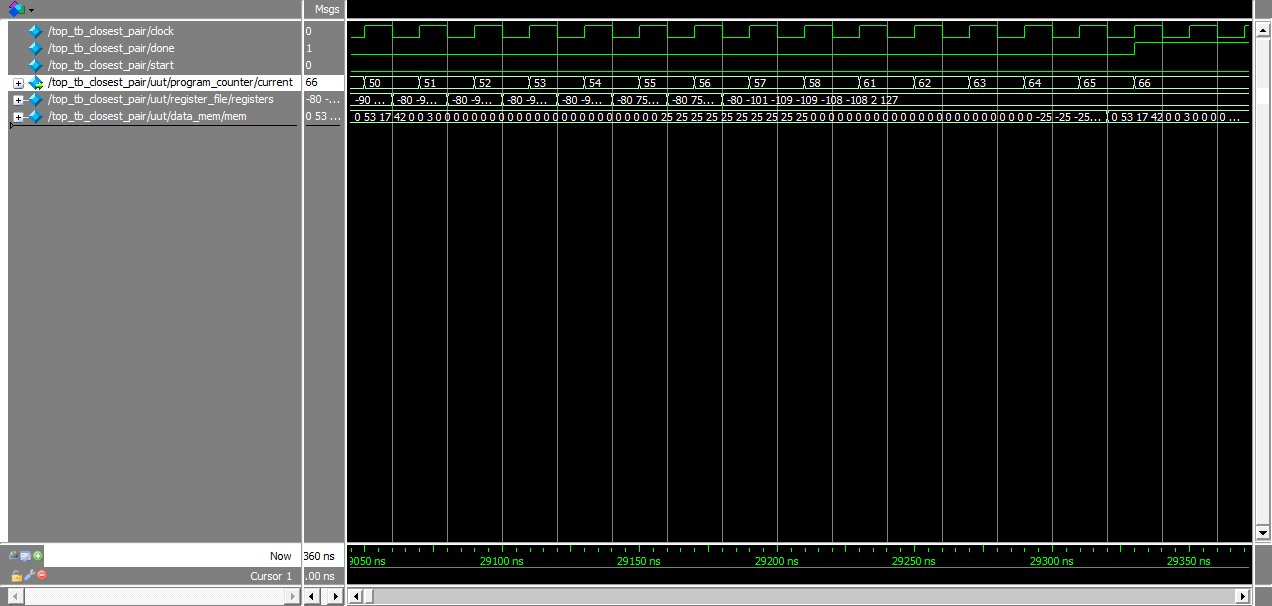
In this view, the program is about to finish so we are seeing the match count about to reach its final value and the last string entries being compared.

Memory: The signal named “…/data\_mem/mem” represents the values in memory. **The values in memory locations 1-3 can be ignored** because they do not affect this problem at all. They are just initialized from the product problem.

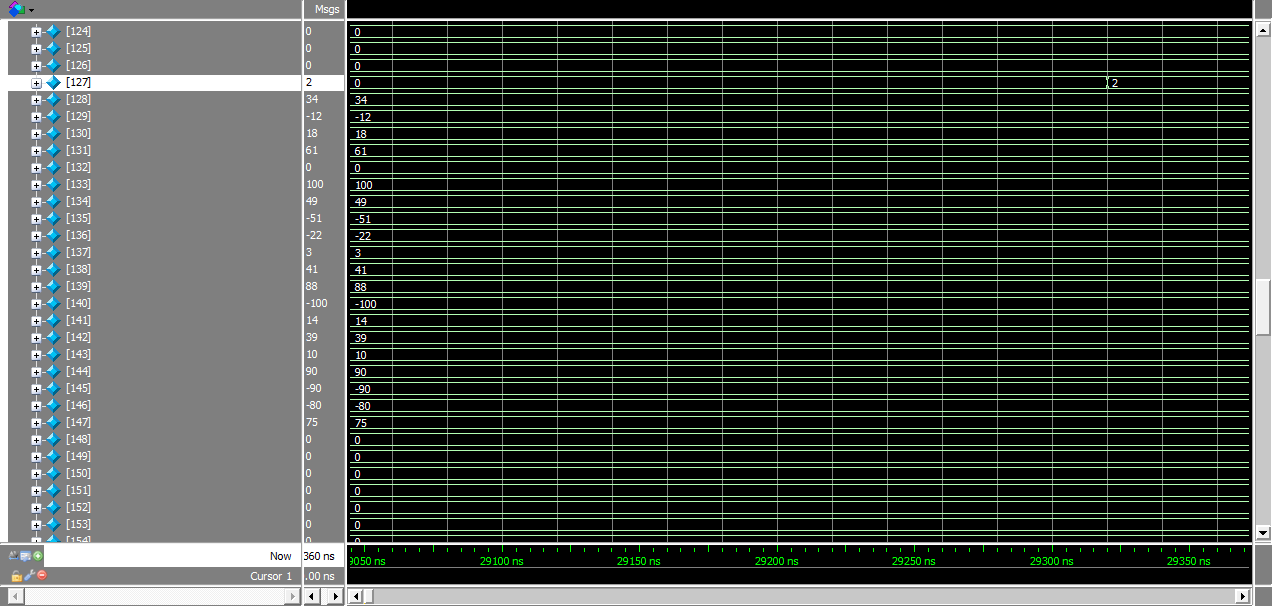
|  |  |
| --- | --- |
| Memory location | What memory location holds |
| 6 | Holds the string pattern that we are searching for |
| 7 | The resulting number of entries in the array that match the string pattern |

In this run of string match we are comparing the string pattern 0011 to an array in which 10 of the entries have value 00011001, 20 have the value 10111101, 9 have value 11100111, and the rest have 00000000. So the correct match count should be 19, which is what our match count is.

**Closet Pair Timing Diagram**



This image is just an overview of the output of the program. The next image shows the relevant data.



Registers The signal named “…/register\_file/registers” represents the current state of the data inside our registers. The values in the registers store the intermediate data of our product algorithm, such as when we calculate the distance. The reason that the values are truncated and not expanded, like memory, is because it would only serve to clutter the timing diagram for this problem.

Memory: The signal named “…/data\_mem/mem” represents the values in memory. Mem[128] – Mem[147] are the values of the array from which we find the minimum distance value. Mem [127] is where the result is stored, which is why the value 2 is stored many cycles later.

In this problem, the array of values that we are calculating the minimum distance value from are 34, -12, 18, 61, 0, 100, 49, -51, -22, 3, 41, 88, -100, 14, 39, 10, 90, -90, -80, and 75. The pair with the least distance is 90 and 88, and the distance is 2. The correct value of 2 is stored in Mem[127].

**Assembly Code of Programs**

**PRODUCT**

load $r3, $r3 ; R[3] = M[R[3]] ($r3 = A)

load $r4, $r4 ; R[4] = M[R[4]] ($r4 = B)

load $r5, $r5 ; R[5] = M[R[5]] ($r5 = C)

; MULTIPLY A AND B

while1:

and $r4 ; $overflow = B & 1

bno if1 ; branch if LSB of B is 0

add $r1, $r3 ; add lower bits

bno if1 ; branch if no overflow

incr $r0 ; increment upper bits

add $r0, $r2 ; add upper bits

if1:

lsl $r2 ; shift upper bits of A left

lsl $r3 ; shift lower bits of A left

bno if2 ; skip if MSB lower bit is 0

incr $r2 ; add overflow to upper bits of A

if2:

lsr $r4 ; shift B right

eqz $r4 ; overflow = (B == 0)

bno while1 ; loop while B != 0

# MULTIPLY AB AND C

zero $r2 ; initialize $r2 to 0

zero $r3 ; initialize $r3 to 0

while2:

and $r5 ; $overflow = C & 1

bno if3 ; branch if LSB of C is 0

add $r3, $r1 ; add lower bits

bno if3 ; branch if no overflow

incr $r2 ; increment upper bits

add $r2, $r0 ; add upper bits

if3:

lsl $r0 ; shift upper bits of AB left

lsl $r1 ; shift lower bits of AB left

bno if4 ; skip if MSB lower bit was 0

incr $r0 ; add overflow to upper bits of AB

if4:

lsr $r5 ; shift C right

eqz $r5 ; overflow = (C == 0)

bno while2 ; loop while C != 0

store $r2, $r6 ; store upper bits in memory

store $r3, $r7 ; store lower bits in memory

halt

**STRING MATCH**

;LOAD VALUES OF OPERANDS

load $r1, $r1 ; load pattern from memory

;OUTER LOOP THROUGH ALL INPUT BYTE STRINGS

s\_while1:

load $r0, $r6 ; load first byte string from memory

;INNER LOOP THROUGH EVERY 4 BIT COMBO IN EACH STRING

s\_while2:

match $r0, $r1 ; check if lower 4 bits of string match pattern

bof s\_if1 ; if pattern match, branch

lsr $r0 ; if no match, s >> 1

incr $r5 ; increment inner loop index, j, by 1

lt $r4, $r5 ; check inner while condition (j < length)

bno s\_if2 ; exit loop

bof while2 ; if (j < length) is true, loop inner

s\_if1:

incr $r2 ; increment count by 1

s\_if2:

incr $r6 ; increment outer loop index, i, by 1

lt $r6, $r7 ; check outer while condition (i < size)

bof s\_while1 ; if (i < size) is true, loop outer

store $r2, $r3 ; store the count to memory

halt

**CLOSEST PAIR**

; OUTER LOOP THROUGH EVERY VALUE IN ARRAY TO COMPARE

c\_while1:

load $r0, $r2 ; x = array[i]

incr $r2 ; i++

zero $r4 ; j = 0

add $r4, $r2 ; j += i (so that j = i)

;INNER LOOP THROUGH EVERY OTHER VALUE IN ARRAY TO COMPARE TO

c\_while2:

load $r1, $r4 ; y = array[j]

incr $r4 ; j++

dist $r1, $r0 ; calculate dist, overwrite y

lt $r1, $r6 ; check if dist < min

bno c\_if1 ; if dist >= min, skip min update

zero $r6 ; min = 0

add $r6, $r1 ; min += dist

c\_if1:

lt $r4, $r5 ; check if j < 20

bof c\_while2 ; if j < 20 is true, then loop inner

lt $r2, $r3 ; check if i < 19

bof c\_while1 ; if i < 19 is true, then loop outer

store $r6, $r7 ; store min in result address

halt

**Machine Code of Programs**

000011011

000100100

000101101

110011100

111000101

010001011

111000011

110010000

010000010

110000010

110000011

111000010

110010010

110001100

110100100

111010100

110101010

110101011

110011101

111000101

010011001

111000011

110010010

010010000

110000000

110000001

111000010

110010000

110001101

110100101

111010100

001010110

001011111

110111000

000001001

000000110

011000001

111100110

110001000

110010101

100100101

111000011

111101000

110010010

110010110

100110111

111110101

001010011

110111000

000000010

110010010

110101100

010100010

000001100

110010100

101001000

100001110

111000011

110101110

010110001

100100101

111111000

100010011

111110010

001110111

110111000

**Questions**

Question 1: Have you made any changes to you ISA from Lab1? What were they? Why?

We did not make any changes to our ISA since Lab1.

Question 2: What are your dynamic instruction counts for programs 1, 2, and 3?

Product: 215

String Match: 2453

Closest Pair: 1825

Question 3: What could you have done differently to better optimize for dynamic instruction count?

We could have made our core algorithms for each problem more efficient. For example, our closest pair algorithm has a nested for loop so it has a worst case of runtime of O(n2). If we would have used a more efficient algorithm, like one that would have taken linear time, then the dynamic instruction count would innately be lower. However, the tradeoff, that we decided against in lab 1, was that a more efficient algorithm would have led to more room for error.

Question 4: Execution time

Product:

Total time: 2390 ns

Cycle time: 20 ns

Total clock cycle count: 120

Fmax = 45.8 MHz

**Execution time**: 2.62 \* 10^-6 s

String Match

Total time: 43750 ns

Cycle time: 20 ns

Total clock cycle count: 2188

Fmax = 45.8 MHz

**Execution time**: 4.78 \* 10^-5 s

Closest Pair

Total time: 29330 ns

Cycle time: 20 ns

Total clock cycle count: 1467

Fmax = 45.8 MHz

**Execution time**: 3.2 \* 10^-5 s