Tristan Seroff

11/11/15

ECE 154A Lab 4

a. Hours spent on lab: Roughly 10

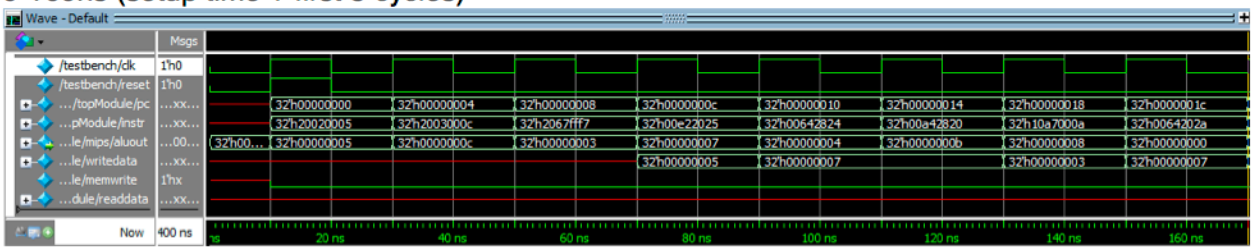
b. Table 1: First sixteen cycles of test code from textbook Fig. 7.60

Cycle	reset	pc	Instr	branch	srca	srcb	aluresult	zero	pcsrc	Writedata	mem write	read data
1	1	00	addi $2, $0, 5 20020005	0	0	5	5	0	0	~~0~~ X	0	X
2	0	04	addi $3, $0, 12 2003000C	0	0	C	C	0	0	~~0~~ X	0	X
3	0	08	addi $7, $3, -9 20067FFF7	0	C	-9	3	0	0	~~0~~ X	0	X
4	0	0C	or $4, $7, $2 00E22025	0	3	5	7	0	0	5	0	X
5	0	10	and $5, $3, $4 00642824	0	12	7	4	0	0	7	0	X
6	0	14	add $5, $5, $4 00A42820	0	4	7	11	0	0	7	0	X
7	0	18	beq $5, $7, end 10A7000A	1	11	3	8	0	0	3	0	X
8	0	1C	slt $4, $3, $4 0064202A	0	12	7	0	1	0	7	0	X
9	0	20	beq $4, $0, around 10800001	1	0	0	0	1	1	0	0	X
10	0	28	slt $4, $7, $2 00E2202A	0	3	5	1	0	0	5	0	X
11	0	2C	add $7, $4, $4 00853820	0	1	11	12	0	0	11	0	X
12	0	30	sub $7, $7, $2 00E23822	0	12	5	7	0	0	5	0	X
13	0	34	sw $7, 68($3) ac670044	0	12	68	80	0	0	7	1	X
14	0	38	lw $2, 80($0) 8C020050	0	0	80	80	0	0	X	0	7
15	0	3C	j end 08000011	0	X	X	XX	X	0	X	0	X
16	0	44	sw $2, 84($0) AC020054	0	0	84	84	0	0	7	1	X

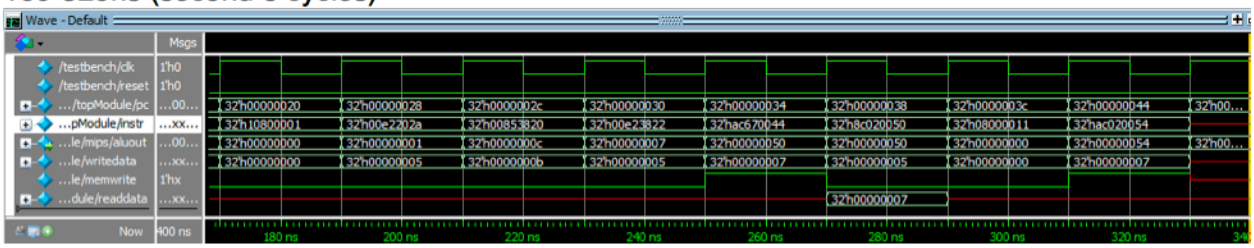
The sw instruction at PC = 0x44 of the program in memfile.dat **writes 0x0000 0007 (7) to memory address 0x54 (84)**

c. Simulation waveforms of original processor:

0-160ns (setup time + first 8 cycles)

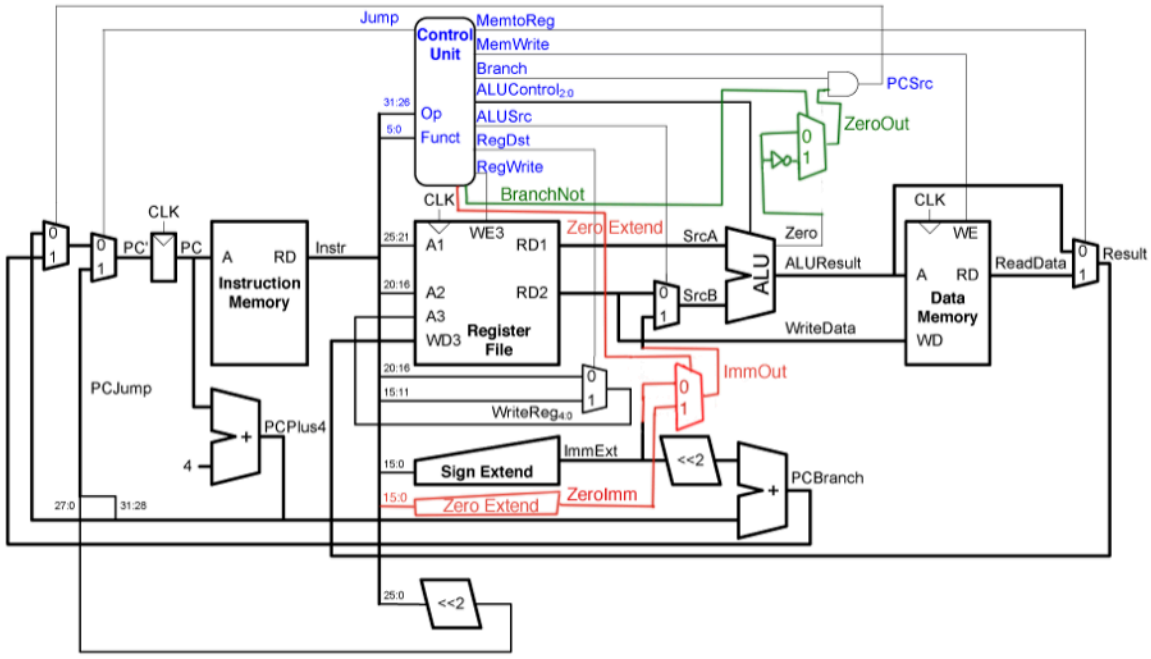


160-320ns (second 8 cycles)



d. Modifications to support ori and bne operations

Modified datapath:



Modified **Single-cycle MIPS processor**

A new signal, BranchNot, should be added such that the input to the AND gate for PCSrc is selected by a 2:1 multiplexer - an input of 0 propagates the unchanged Zero signal from the ALU, used for the beq operation, while an input of 1 instead connects the Zero signal through a NOT gate, to be used for the bne operation.

A new signal, ZeroExtend, should be added such that the input to port 1 of the SrcB multiplexer to the ALU is selected by a 2:1 multiplexer - an input of 0 propagates ImmExt, the sign-extended immediate, used for existing operations, while an input of 1 instead connects a zero-extended version of the immediate, used for the ori operation.

See section F for modified main decoder table, Table 2
Modified ALU decoder table:

$ALUOp_{1:0}$	Meaning	$Funct_{5:0}$	Meaning	$ALUControl_{2:0}$
00	Add	X	Add	010
01	Subtract	X	Subtract	110
10	Look at funct field	100000	Add (R-type)	010
		100010	Subtract (R-type)	110
		100100	AND (R-type)	000
		100101	OR (R-type)	001
		101010	SLT (R-type)	111
11	Or	X	OR (I-type)	001

For the bne operation, all existing control signals are the same as those of the beq operation. The new signal, BranchNot, is 1 for bne, and 0 for all other operations.

To support the ori operation, the main decoder should output 11 for $APUOp_{1:0}$, and the ALU decoder should output 001 is response to this signal. The new signal, ZeroExtend, should be 1 for ori and 0 for all other existing operations. All other control signals for ori should be the same as those of the addi operation.

e. Verilog code for the modified MIPS processor

Testbench.v

```
// Testbench for MIPS processor
// Issues reset signal then cycles clock repeatedly
module tb;
reg clk;
reg reset;
top topModule(clk, reset); // Instantiate MIPS top module
reg i;

initial begin
      clk = 0;
      reset = 0;
      #10;
      reset = 1;
      clk = 1;
      #10
      reset = 0;
      clk = 0;
      for (i = 0; i < 16; i = i + 1) begin
            // Cycle clock
            #10
            clk = 1;
            #10
            clk = 0;
      end
end

endmodule
```

mipstop.v:

```
// Top level system including MIPS and memories
// Unmodified from provided code

module top(input clk, reset);

  wire [31:0] pc, instr, readdata;
  wire [31:0] writedata, dataadr;
  wire        memwrite;

  // processor and memories are instantiated here
  mips mips(clk, reset, pc, instr, memwrite, dataadr, writedata,
readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);

endmodule
```

mipsmem.v:

```
// External memories used by MIPS single-cycle processor

// Data memory implementation
module dmem(input          clk, we,
            input   [31:0] a, wd,
            output  [31:0] rd);

// 64 bit storage of 32-bit words
reg [31:0] RAM[63:0];

// Always read
assign rd = RAM[a[31:2]];

always @(posedge clk) begin
      // Write if enabled
      if (we) begin
            RAM[a[31:2]] <= wd;
      end
end

endmodule


// Instruction memory (already implemented)
module imem(input   [5:0]  a,
            output  [31:0] rd);

  reg [31:0] RAM[63:0];

  initial
    begin
      $readmemh("memfile2.dat",RAM); // Initialize memory with program
    end

  assign rd = RAM[a]; // word aligned
endmodule
```

mips.v:

```
// Single-cycle MIPS processor
// Instantiates a controller and a datapath module

module mips(input          clk, reset,
            output  [31:0] pc,
            input   [31:0] instr,
            output         memwrite,
            output  [31:0] aluout, writedata,
            input   [31:0] readdata);

  wire        memtoreg, branch,
              pcsrc, zero,
              alusrc, regdst, regwrite, jump;
  wire [2:0]  alucontrol;

  // ADDED: wire for zeroExtend signal
  wire zeroExtend;

  // ADDED: zeroExtend output from controller, input to datapath
  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol, zeroExtend);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata, zeroExtend);
endmodule


// Controller module
module controller(input   [5:0] op, funct,
                  input         zero,
                  output        memtoreg, memwrite,
                  output        pcsrc, alusrc,
                  output        regdst, regwrite,
                  output        jump,
                  output  [2:0] alucontrol,
                  output zeroExtend); // ADDED: ZeroExtend output to
datapath

wire [1:0] aluop;
wire branch;
wire branchNot; // ADDED: branchNot signal for bne operation
```

```
// ADDED: zeroExtend output from main decoder
mainDecoder dec(op, memtoreg, memwrite, branch, alusrc, regdst,
regwrite, jump, aluop, branchNot, zeroExtend);
aluDecoder aluDec(funct, aluop, alucontrol);

// Modification for branchNot here
wire zeroOut;
wire zeroNot;
not zeroNotGate(zeroNot, zero);
mux2 #(1) branchMux(zero, zeroNot, branchNot, zeroOut);
assign pcsrc = branch & zeroOut; // Changed from branch & zero

endmodule

// Datapath
module datapath(input          clk, reset,
                input          memtoreg, pcsrc,
                input          alusrc, regdst,
                input          regwrite, jump,
                input   [2:0]  alucontrol,
                output         zero,
                output  [31:0] pc,
                input   [31:0] instr,
                output  [31:0] aluout, writedata,
                input   [31:0] readdata,
                input zeroExtend);    // ADDED: zeroExtend input from
controller

wire [4:0] writereg;
wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
wire [31:0] signimm, signimmsh;
wire [31:0] srca, srcb;
wire [31:0] result;

// Determine next PC
reset_ff #(32) pcreg(clk, reset, pcnext, pc);
adder pcadd1(pc, 32'b100, pcplus4);
sl2 immsh(signimm, signimmsh);
adder pcadd2(pcplus4, signimmsh, pcbranch);
mux2 #(32) pcbrmuc(pcplus4, pcbranch, pcsrc, pcnextbr);
wire [31:0] pcmuxd0;
assign pcmuxd0[31:0] = {pcplus4[31:28], instr[25:0], 2'b00};
mux2 #(32) pcmux(pcnextbr, pcmuxd0, jump, pcnext);

// Register file
regfile rf(clk, regwrite, instr[25:21], instr[20:16], writereg,
result, srca, writedata);
mux2 #(5) wrmux(instr[20:16], instr[15:11], regdst, writereg);
mux2 #(32) resmux(aluout, readdata, memtoreg, result);
```

```
signext se(instr[15:0], signimm);

// ADDED: logic for zeroExtend
wire [31:0] zeroImm;
zeroext zext(instr[15:0], zeroImm);
wire [31:0] immOut;
mux2 #(32) immMux(signimm, zeroImm, zeroExtend, immOut);

// ALU
// ADDED: changed signimm to immOut to get signal from mux
mux2 #(32) srcbmux(writedata, immOut, alusrc, srcb);
alu alu(srca, srcb, alucontrol, aluout, zero);

endmodule
```

controller.v:

```
module mainDecoder(   input [5:0] op,
                      output reg memtoreg,
                      output reg memwrite,
                      output reg branch,
                      output reg alusrc,
                      output reg regdst,
                      output reg regwrite,
                      output reg jump,
                      output reg [1:0] aluop,
                      output reg branchNot,
                      output reg zeroExtend);

// Main decoder
reg [10:0] controls; // ADDED two bits for branchNot and zeroExtend

always @ (*) begin
      case(op)
            // Controls register modified to include branchNot,
zeroExtend signals
            6'b000000: controls <= 11'b00110000010; // R-type
            6'b100011: controls <= 11'b00101001000; // lw
            6'b101011: controls <= 11'b00001010000; // sw
            6'b000100: controls <= 11'b00000100001; // beq
            6'b001000: controls <= 11'b00101000000; // addi
            6'b000010: controls <= 11'b00000000100; // j
            6'b001101: controls <= 11'b10101000011; // ADDED: ori
            6'b000101: controls <= 11'b01000100001; // ADDED: bne
            default: controls <= 11'bxxxxxxxxxxx; // invalid opcode
      endcase
      zeroExtend = controls[10]; // ADDED: zeroExtend
      branchNot = controls[9]; // ADDED: branchNot
      regwrite = controls[8];
      regdst = controls[7];
      alusrc = controls[6];
      branch = controls[5];
      memwrite = controls[4];
      memtoreg = controls[3];
      jump = controls[2];
      aluop = controls[1:0];
end
endmodule

module aluDecoder(    input [5:0] funct,
                      input [1:0] aluop,
                      output reg [2:0] alucontrol);
// ALU decoder
always @ (*) begin
      case (aluop)
```

```
            2'b00: alucontrol <= 3'b010; // add for lw, sw, addi
            2'b01: alucontrol <= 3'b110; // sub for beq
            2'b11: alucontrol <= 3'b001; // ADDED: or for ori operation
            default: case (funct)      // R-type functions
                  6'b100000: alucontrol <= 3'b010; // add
                  6'b100010: alucontrol <= 3'b110; // sub
                  6'b100100: alucontrol <= 3'b000; // and
                  6'b100101: alucontrol <= 3'b001; // or
                  6'b101010: alucontrol <= 3'b111; // slt
                  default: alucontrol <= 3'bxxx; // invalid data in
funct field
            endcase
      endcase
end
endmodule
```

datapath.v:

```
// Datapath modules

// Register file
module regfile(  input reg clk,
           input reg we3,
           input reg [4:0] ra1, ra2, wa3,
           input reg [31:0] wd3,
           output [31:0] rd1, rd2);
reg [31:0] rf [31:0];

always @ (posedge clk) begin
     if (we3) begin
          rf[wa3] <= wd3;
     end
end

assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

// Adder
module adder(    input reg [31:0] a, b,
           output [31:0] y);
assign y = a + b;
endmodule

// Bitshift left by 2
module sl2(      input reg [31:0] a,
           output [31:0] y);
assign y = {a[29:0], 2'b00};
endmodule

// Sign extension
module signext(  input reg [15:0] a,
           output [31:0] y);
assign y = {{16{a[15]}}, a};
endmodule

// Variable-width resettable flip-flop
module reset_ff #(parameter WIDTH = 8)(
           input reg clk, reset,
           input reg [(WIDTH-1):0] d,
           output reg [(WIDTH-1):0] q);
always @ (posedge clk, posedge reset)
     if (reset) begin
          q <= 0;
     end
     else begin
```

```
            q <= d;
      end
endmodule

// Variable-width 2:1 multiplexer
module mux2 #(parameter WIDTH = 8)(
            input reg [(WIDTH-1):0] d0,
            input reg [(WIDTH-1):0] d1,
            input reg s,
            output [(WIDTH-1):0] y);
assign y = s ? d1 : d0;
endmodule

// ADDED: zero extension module for ori operation
module zeroext(  input reg [15:0] a,
            output [31:0] y);
assign y = {16'b0000000000000000, a};
endmodule
```

alu.v:

```
// ALU from Lab 1 - unmodified
module alu(input [31:0] a, b, input [2:0] f, output [31:0] y, output
zero);

reg [31:0] addB; // signal fed to full adder
wire [31:0] addY;     // number from full adder
wire [31:0] aandb;    // A & (~)B
wire [31:0] aorb;     // A | (~)B
wire [31:0] asltb;    // A SLT B

fullAdder FA(a, addB, f[2], addY);

// Bitwise operations
assign aandb = (a & addB);
assign aorb = (a | addB);

// SLT from sign bit of adder result, zero extend
assign asltb[31:1] = 31'b0000000000000000000000000000000;
assign asltb[0] = (addY[31] == 1);

// Buffer for output
reg[31:0] y_out;
assign y = y_out;

// Maps B/~B as appropriate, output to correct function result
always @ * begin
case(f[2])
      0: addB = b;
      1: addB = ~b;
      default: addB = 32'h00000000;
endcase
case(f)
      0: y_out = aandb;
      1: y_out = aorb;
      2: y_out = addY;
      4: y_out = aandb;
      5: y_out = aorb;
      6: y_out = addY;
      7: y_out = asltb;
      default: y_out = 32'h00000000;
endcase
end
assign zero = (y == 32'h00000000); // If output is zero, toggle zero
endmodule

module fullAdder(input [31:0] a, b, input cin, output [31:0] y);
      assign y = a + b + cin;
endmodule
```

f. Tables 2 and 3 for the modified MIPS processor:

Table 2: Extended functionality for the main decoder

Instruction	$Op_{5:0}$	Reg Write	RegDst	AluSrc	Branch	MemWrite	Memto Reg	ALU $Op_{1:0}$	Jump	Branch Not	ZeroExtend
R-type	000000	1	1	0	0	0	0	10	0	0	0
lw	100011	1	0	1	0	0	1	00	0	0	0
sw	101011	0	X	1	0	1	X	00	0	0	0
beq	000100	0	X	0	1	0	X	01	0	0	0
addi	001000	1	0	1	0	0	0	00	0	0	0
j	000010	0	X	X	X	0	X	XX	1	0	0
ori	001101	1	0	1	0	0	0	11	0	0	1
bne	000101	0	X	0	1	0	X	01	0	1	0

Table 3: Extended functionality for the ALU decoder

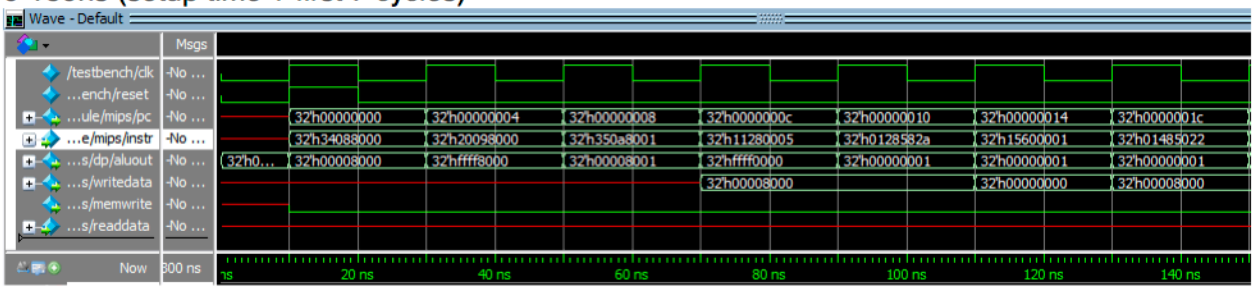
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at funct field
11	OR

g. Contents of memfile2.dat, machine code translation of test2.asm:

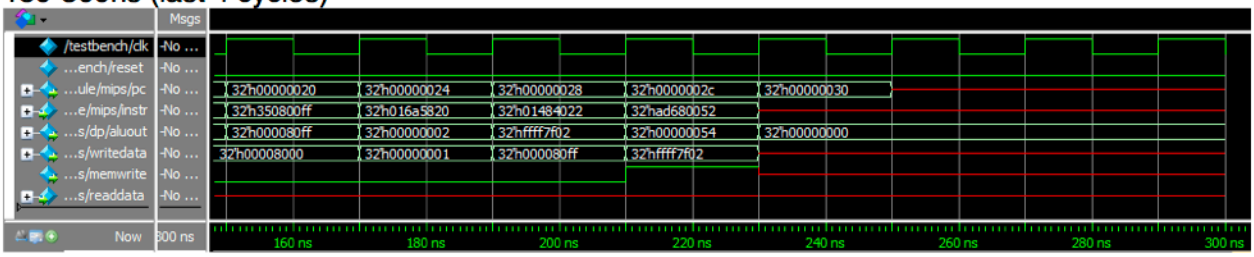
```
34088000
20098000
350A8001
11280005
0128582A
15600001
08000005
01485022
350800FF
016A5820
01484022
AD680052
```

h. Simulation waveforms of modified processor

0-150ns (setup time + first 7 cycles)



150-300ns (last 4 cycles)



The sw instruction at PC 0x30 of test2.asm **writes 0xFFFF 7F02 (-33022) to memory address 0x54 (84)**

Addendum: Translation of test2.asm into machine code

```
main: (PC = 0x00)
      ori $t0 $0, 0x8000
```

op	rs (source)	rt (source/destination)	imm
13	0	8	0x8000
0011 01	00 000	0 1000	1000 0000 0000 0000

`0x34088000`

```
      addi $t1, $0, -32768
```

op	rs (source)	rt (source/destination)	imm
8	0	9	-32768
0010 00	00 000	0 1001	1000 0000 0000 0000

`0x20098000`

```
      ori $t2, $t0, 0x8001
```

op	rs (source)	rt (source/destination)	imm
13	8	10	0x8001
0011 01	01 000	0 1010	1000 0000 0000 0001

`0x350A8001`

```
      beq $t0, $t1, there (PC = 0x0C)
```

op	rs (source)	rt (source/destination)	imm
4	9	8	BTA = 0x24 SignImm = 0x24 - 0x10 = 0x14 >> 2 = 5
0001 00	01 001	0 1000	0000 0000 0000 0101

`0x11280005`

```
      slt $t3, $t1, $t0
```

op	rs (source)	rt (source)	rd (destination)	shamt	funct
0	9	8	11	0	42
0000 00	01 001	0 1000	0101 1	000 00	10 1010

`0x0128582A`

```
      bne $t3, $0, here (PC = 0x14)
```

op	rs (source)	rt (source/destination)	imm
5	11	0	BTA = 0x1C SignImm = 0x1C - 4 - 0x14 = 0x08 >> 2 = 1
0001 01	01 011	0 0000	0000 0000 0000 0001

```
0x15600001
```

```
      j there (PC = 0x18)
```

op	address
2	JTA = 0x24 0000 0000 0010 0100
0000 10	00 0000 0000 0000 0000 0000 0101

```
0x08000005
```

```
here: (PC = 0x1C)
      sub $t2, $t2, $t0
```

op	rs (source)	rt (source)	rd (destination)	shamt	funct
0	10	8	10	0	34
0000 00	01 010	0 1000	0101 0	000 00	10 0010

```
0x01485022
```

```
      ori $t0, $t0, 0xFF
```

op	rs (source)	rt (source/destination)	imm
13	8	8	0xFF
0011 01	01 000	0 1000	0000 0000 1111 1111

```
0x350800FF
```

```
there: (PC = 0x24)
      add $t3, $t3, $t2
```

op	rs (source)	rt (source)	rd (destination)	shamt	funct
0	11	10	11	0	32
0000 00	01 011	0 1010	0101 1	000 00	10 0000

```
0x016A5820
```

`sub $t0, $t2, $t0`

op	rs (source)	rt (source)	rd (destination)	shamt	funct
0	10	8	8	0	34
0000 00	01 010	0 1000	0100 0	000 00	10 0010

`0x01484022`

`sw $t0, 82($t3)`

op	rs (source)	rt (source/destination)	imm
43	11	8	82
1010 11	01 011	0 1000	0000 0000 0101 0010

`0xAD680052`

`Ideal Program operation:`

PC	Instruction performed
0x00	`$t0 = 0x0000 8000`
0x04	`$t1 = 0xffff 8000`
0x08	`$t2 = 0x0000 8001`
0x0C	`branch if $t0 and $t1 are equal - they're not`
0x10	`$t3 = #t1 < $t0 = 1`
0x14	`branch if $t3 ≠ 0, yes, go to 'here' => PC = 0x1C`
0x18	`Not executed`
0x1C	`$t2 = 0x0000 0001`
0x20	`$t0 = 0x0000 80FF (33023)`
0x24	`$t3 = $t3 + $t2 = 1 + 1 = 2`
0x28	`$t0 = $t2 - $t0 => 1 - 33023 => -33022 => 0xFFFF7F02`
0x2C	`$t0 saved to $t3 + 82 => -33022 stored in 84`