Instruction Decode (ID) Pipeline Stage - Team D

Cherukuri,Sampath Kumar
Panchmahalkar,Pritheesh
Sharp,Elena E
Wan,Yao
Zeng,Qiliang
Zou,Diyang

CSE 5/7381
Final Report For December 9th, 2017

General Project Description

B. Instruction Decode (ID) Pipeline Stage Using Verilog, develop the Instruction Decode (ID) stage of the MIPS pipeline. This will include development of the register file and other related components. You will use the Fibonacci machine code for testing this stage. You will need to develop supporting hardware and software to test the operation of this stage, including:

1. ID – decode instructions, read/write register file
    a. Develop register file and decoder
    b. Decode instructions (R-type, I-type, J-type)
    c. Read and write from registers in register file

Components: Registers, Sign-extend, MUX input to Write register on Registers
Control signals: RegDst, RegWrite

Team Duties

Project Manager – will manage the project and make sure that the team is following the schedule to complete all required project tasks. All communication with “upper management” (Dr. Manikas) regarding the project is to be performed by the project manager.

Technical Writer – develops report drafts, including final drafts for submission

Hardware Designer(s) – develops Verilog code and component designs

Software Designer(s) – develops MIPS code to run on hardware

Test Engineer(s) – develops testbench and test procedures to verify design functionality (hardware and software)

Project Manager	Sampath Cherukuri
Technical Writer	Elena Sharp
Hardware Design(s)	Pritheesh Panchmahalkar, Diyang Zou, Zeng Qiliang
Software Design(s)	Elena Sharp
Test Engineer(s)	Sampath Cherukuri, Yao Wan, Pritheesh Panchmahalkar

Design Implementation and Testing & Code Files and Implementation

Testing Technologies

The compiler we used to test is the Icarus Verilog which is a Verilog simulation and synthesis tool. It is much more easier than the terminal in Mac because it shows the error position and reason which helps solve the problems during coding and testing. Using the command iverilog can produce a .exe file which can launched with a window of results like the XQuartz work bench. It is also easy to test multiple test benches, because all we have to do is to modify the test bench and the name of .exe file , not to new a test bench and then run the .exe files with different names means different tests.

1. Register File

Original Design

The register size of the MIPS basic instruction format is 5 bits, and the instruction field can access 2^5 = 32 32-bit registers. Such a stack of registers make up the register file.

1.1 Process Description

Step 1: Reset works using the flag, using i to count, when i <31, flag is equal to 0; until i = 32, reset is completed, flag = 1, Then, registers can be written.

Step 2: During reset, need 32 pulses to reset all the registers. When the reset is not completed, the flag is always equal to 0 and the registers can not be written. If executing the write then go to error and then divide by 32, when the register can be written after that the reset is complete.

1.2 Register File Interface Description

pulse	clk,
write enable	reg_write,
write address	wr_addr,
write data	wr_data,
read address 1	r1_addr,
read address 2	r2_addr,
read data 1	r1_data,
read data 2	r2_data;

Implementation

To create the register file, we first had to look at the different parts of the instructions, and how those instructions were going to use the register file. As shown in our design above, we needed to account for instructions that just wanted to read from the registers as well as instructions that were looking to write into the register file.

As seen in I-type instructions, we would need to read from at most two registers per instruction, so in order to account for this, we allow the user to pass in the address of the first read (r1_addr), as well as place to store the data once accessed (r1_data), and the address and data for the second read (r2_addr and r2_data).
For instructions that wish to write to the register file, they will pass in the address of the write address (wr_addr) and the data they wish to write to that register (wr_data).

The last two inputs that we have are for the clock (clk) and the write enable signal (reg_write). How the module works is as follows: the first time the module is used, all the variables will be initialized, and the register file will be initialised to zero. With each call, this module will grab the values for r1_data and r2_data, and then on the rising edge of the clock and if the reg_write signal is set, wr_data will be written into the register file.

Code Listing:

```
1   // This module implements the register file
2
3
4   module register_file(
5     clk,
6     r1_addr,
7     r1_data,
8     r2_addr,
9     r2_data,
10    wr_addr,
11    wr_data,
12    reg_write
13    );
14
15    input clk;                          // input clock
16    input [4:0]  r1_addr;               // input read register A address
17    input [4:0]  r2_addr;               // input read register B address
18    input [4:0]  wr_addr;               // input write register address
19    input [31:0] wr_data;               // input write data
20    input reg_write;                    // input RegWrite signal
21
22    output [31:0] r1_data;              // output read register A data
23    output [31:0] r2_data;              // output read register B data
24
25    reg [31:0] reg_file [31:0];       // declaring 32 registers with size 32 bits
26
27    assign r1_data = reg_file[r1_addr];   // assign data in A's address of
28                                          // reg_file to output read register A
29    assign r2_data = reg_file[r2_addr];   // assign data in B's address of
30                                          // reg_file to output read register B
31
32    integer i;
33
```

```
34    initial begin    // begin initial block to initialise the registers to 0
35      for(i = 0; i < 32; i = i+1) begin
36        reg_file[i] = 0;
37      end
38    end
39
40    // sets the write data whenever there is rising edge on clock and when
41    // RegWrite signal is set
42    always @(posedge clk) begin
43      if (reg_write) begin
44        reg_file[wr_addr] <= wr_data;
45      end
46    end
47
48  endmodule
49
```

Testing:

To test that the register file we wrote was functioning as expected, we used two different tests for function test. The first test was using the module to write a number to the register file and ensure that it was properly storing the correct number in the correct place. The second part of our test bench was written to ensure that the write enable signal was functioning correctly, and that the register file was only being written to when that signal was set. This result is based on that the priority of the write enable signal follows the clk posedge.

Code:

```
1   // testbench for register_file.v
2   module register_file_tb;
3       reg clk;
4       reg [4:0]  r1_addr;
5       reg [4:0]  r2_addr;
6       reg [4:0]  wr_addr;
7       reg [31:0] wr_data;
8       reg reg_write;
9
10      wire [31:0] r1_data;
11      wire [31:0] r2_data;
12
13      register_file test(clk,r1_addr,r1_data,r2_addr,r2_data,wr_addr,wr_data,reg_write);
14
15      // Test Case 1:
16      // Write '42' to register 2, verify with Read 1 and 2
17      initial
18      begin
19          wr_addr = 5'd2;
20          wr_data = 32'd42;
21          reg_write = 1;
22          r1_addr = 5'd2;
23          r2_addr = 5'd2;
24          #1 clk=1; #1 clk=0; // Generate single clock pulse
25          // Verify expectations and report test result
26          if((r1_data != 42) || (r2_data != 42))
27              $display("Test Case 1 Failed");
28          else
29              $display("Test Case 1 Succeed");
30          $display($time," reg_write=%b",reg_write);
31          $display($time," r1_data=%d,r2_data=%d,wr_data=%d",r1_data,r2_data,wr_data);
32      end
33
```

```
34      // Test Case 2:
35      // Write '41' to register 2, verify write enable signal
36      initial
37      begin
38          #2 wr_addr = 5'd2;
39          #2 wr_data = 32'd41;
40          #2 reg_write = 0;
41          #2 r1_addr = 5'd2;
42          #2 r2_addr = 5'd2;
43          // Verify expectations and report test result
44          if((r1_data != 41) && (r2_data != 41))
45              $display("Test Case 2 Succeed");
46          else
47              $display("Test Case 2 Failed");
48          $display($time," reg_write=%b",reg_write);
49          $display($time," r1_data=%d,r2_data=%d,wr_data=%d",r1_data,r2_data,wr_data);
50      end
51
52  endmodule
53
```

Test Output: "verilog +gui register_file.v register_file_tb.v &"

```
Last login: Wed Dec  6 20:08:59 on ttys001
Yaos-MacBook-Air:~ yao$ /Users/yao/Desktop/cap/rf ; exit;
Test Case 1 Succeed
                   2 reg_write=1
                   2 r1_data=        42,r2_data=        42,wr_data=        42
Test Case 2 Succeed
                  10 reg_write=0
                  10 r1_data=        42,r2_data=        42,wr_data=        41
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```

As we can see above, the testbench shows that the register file is working as intended.

2. Decode Instruction & Control Signals - RegDst, RegWrite

Original Design

Decode Instruction

Once the instruction register (IR) sends the data, the CPU need the decode this data by checking the correctness, analyzing the data format type, decoding and determining what instruction is to be performed.

2.1 **Instruction Types**

This part including the basic instruction format such as I-format, R-format and J-format.

2.1.1 **MIPS I-format Instruction (**bne, bnq, lw, sw, etc.**)**

I-format:

op (6 bits)	rs (5 bits)	rt (5 bits)	constant or address (16 bits)

Instruction fields:

rs	Destination or source register number
rt	Destination or source register number
constant	-2^{15} to $+2^{15}-1$
address	Offset added to base address in rs

2.1.2 **MIPS R-format Instruction (**add, sub, and, sll, etc.**)**

R-format:

op (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	shamt (5 bits)	funct(6 bits)

Instruction fields:

op	Operation code (opcode)
rs	First source register number

rt	Second source register number
rd	Destination register number
shamt	Shift amount
funct	Function code (extends opcode)

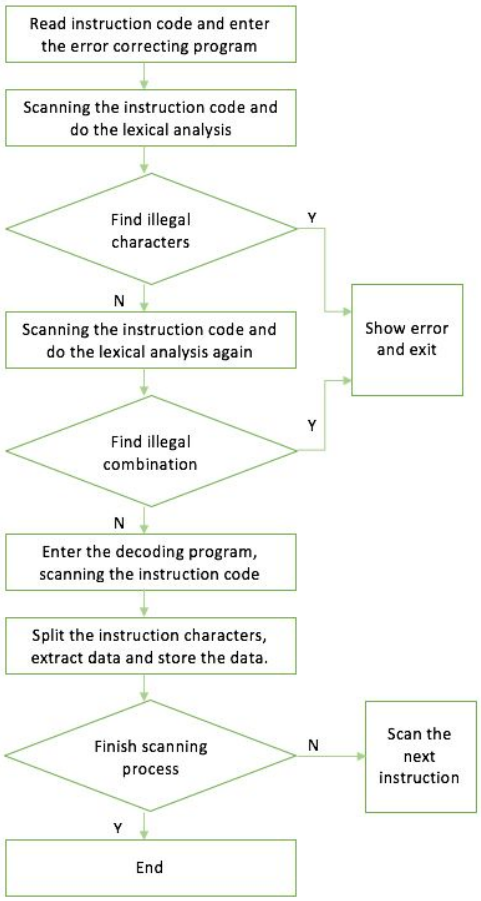
2.1.3 **MIPS J-format Instruction (**j and jal**)**

J-format (Encode full address in instruction):

op (6 bits)	address(26 bits)

2.2 **Decode Process**

The decoding process shows as below:



Control Signals - RegDst, RegWrite

ID produces the control signals for memory and datapath. The inputs are the opcode and functions field of the instruction to control circuit. It generates the MIPS Architecture following kinds of the control signals and what are they:

1. RegWrite
2. AluSrc
3. MemWrite
4. ALUOp
5. MemtoReg
6. MemRead
7. Branch
8. RegDst

The control unit input is the 32-bit instruction word. There are four types of instruction path in control. They are:

- ➔ R-type instruction path:
  - ◆ The R type instructions include ADD, SUB, AND, OR, and SLT.
  - ◆ The ALUOp is determined by the instruction's function field.
- ➔ Lw instruction path:
  - ◆ An example load instruction is lw $t0, 4($sp).
- ➔ Sw instruction path:
  - ◆ An example store instruction is sw $a0, 16($sp).
- ➔ Beq instruction path:
  - ◆ Branch instruction is beq $at, $0, offset.

RegWrite: Some instructions (such as branches, jumps, and stores) do not write to a register.

- ➔ rd field= R-type instruction
- ➔ rt field= I-type instruction
- ➔ $ra (jal instructions)
- ➔ Asserted if a result needs to be written to a register.

RegDst: Select the register destination as either "rd" or "rt"

Control Signal Table

Example:

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemtoReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0

or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

Implementation

When we originally designed these two parts, we had designed them as separate modules. However, we felt that it made more sense to implement the two in the same module as they both use the same parts, and to separate them felt redundant.

We pass a lot of different parts into the decoder module. First we pass the instruction that is going to be decoded (instr) and the clock (clk) in order to keep the system synchronous. Of course op will store the opcode of the decoded instruction, rs, rt, and rd will store the respective registers, shamt will hold the shift amount, funct will store the function code, imm will store the immediate value, target will store the address of the target for a branch. All of these values will be used inside the decoder to decode the instruction. The final results will be stored in RegA, RegB, Imm, and Target. The last two outputs to our code are the control signals RegDst and RegWrite. Once the instruction has been decoded, RegDst and RegWrite will be set to the correct values.

Though the code is lengthy, the decode process is simple. First, we look at the bits 31-26 to get the opcode. We then use the opcode in a case statement to iterate through all the instructions until we find one that matches. Once a match is found, RegA, RegB, Imm, Target, RegDst, and RegWrite are set and returned.

Code Listing:

```
module decoder(instr, clk, rest, op, rs, rt, rd, shamt, funct, imm, target, RegA, RegB, Imm,RegDst, RegWrite,Target);

        input clk;
        input rest;
        input [31:0] instr;

        output [5:0] op;
        output [4:0] rs;
        output [4:0] rt;
        output [4:0] rd;
        output [4:0] shamt;
        output [5:0] funct;
        output [15:0] imm;
        output [25:0] target;
        output [4:0] RegA, RegB;
        output [15:0] Imm;
        output [25:0] Target;
        output RegDst, RegWrite;

        reg [5:0] op;
        reg [4:0] rs;
        reg [4:0] rt;
        reg [4:0] rd;
        reg [4:0] shamt;
        reg [5:0] funct;
        reg [15:0] imm;
        reg [25:0] target;

        reg RegDst, RegWrite;
        reg [4:0] RegA, RegB;
        reg [15:0] Imm;
        reg [25:0] Target;


        always @ (instr)
        begin
                op = instr[31:26];
                rs = instr[25:21];
                rt = instr[20:16];
                rd = instr[15:11];
                shamt = instr[10:6];
                funct = instr[5:0];
                imm = instr[15:0];
                target = instr[25:0];

                case(op)
                // R
                        6'b000000:
                                begin
                                rs = instr[25:21];
                                rt = instr[20:16];
                                rd = instr[15:11];
                                shamt = instr[10:6];
                                funct = instr[5:0];

                                if (funct == 6'b100001) //add
                                        RegA = rs; RegB = rt; RegDst = 1; RegWrite = 1;
                                if (funct == 6'b100000) // add.d
                                        RegA = rs; RegB = rt; RegDst = 1; RegWrite = 1;
                                if (funct == 6'b100100) // and
                                        RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
                                if (funct == 6'b001101) // break
                                        RegA = 5'b00000; RegB = 5'b00000;
                                if (funct == 6'b011010) // div
                                        RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
                                if (funct == 6'b011011) // divu
                                        RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
                                if (funct == 6'b001001) // jalr
                                        RegA = rs; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
```

```
69                          if (funct == 6'b001000) // jr
70                                  RegA = rs; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
71                          if (funct == 6'b010000) // mfhi
72                                  RegA = 5'b00000; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
73                          if (funct == 6'b010010) // mflo
74                                  RegA = 5'b00000; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
75                          if (funct == 6'b010001) // mthi
76                                  RegA = rs; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
77                          if (funct == 6'b010011) // mtlo
78                                  RegA = rs; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
79                          if (funct == 6'b011000) // mult
80                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
81                          if (funct == 6'b011000) // multu
82                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
83                          if (funct == 6'b100111) // nor
84                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
85                          if (funct == 6'b100101) // or
86                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
87                          if (funct == 6'b000000) // sll
88                                  RegA = 5'b00000; RegB = rt; RegWrite = 1; RegDst = 1;
89                          if (funct == 6'b000100) // sllv
90                                  RegA = rt; RegB = rs; RegWrite = 1; RegDst = 1;
91                          if (funct == 6'b101010) // slt
92                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
93                          if (funct == 6'b101011) // sltu
94                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
95                          if (funct == 6'b000011) // sra
96                                  RegA = 5'b00000; RegB = rt; RegWrite = 1; RegDst = 1;
97                          if (funct == 6'b000111) // srav
98                                  RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
99                          if (funct == 6'b000010) // srl
100                                 RegA = 5'b00000; RegB = rt; RegWrite = 1; RegDst = 1;
101                         if (funct == 6'b000110) // srlv
102                                 RegA = rt; RegB = rs; RegWrite = 1; RegDst = 1;
```

```
103                             if (funct == 6'b100010) // sub
104                                     RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
105                             if (funct == 6'b100011) // subu
106                                     RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
107                             if (funct == 6'b001100) // syscall
108                                     RegA = 5'b00000; RegB = 5'b00000; RegWrite = 1; RegDst = 1;
109                             if (funct == 6'b100110) // xor
110                                     RegA = rs; RegB = rt; RegWrite = 1; RegDst = 1;
111                     end
112             //I
113                     6'b001000:  begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// addi
114                     6'b001001:  begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// addiu
115                     6'b001100:  begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// andi
116                     6'b000100:  begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 0; Imm = imm; end// beq
117
118                     6'b000001: begin RegA = rs; RegB = 5'b00001; RegDst = 0; RegWrite = 1; Imm = imm; end// bgez
119                     6'b000111: begin RegA = rs; RegB = 5'b00000; RegDst = 0; RegWrite = 1; Imm = imm; end // bgtz
120                     6'b000110: begin RegA = rs; RegB = 5'b00000; RegDst = 0; RegWrite = 1; Imm = imm; end// blez
121                     6'b000001: begin RegA = rs; RegB = 5'b00000; RegDst = 0; RegWrite = 1; Imm = imm; end// bltz
122
123                     6'b000101: begin RegA = rt; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// bne
124                     6'b100000: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lb
125                     6'b100100: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lbu
126                     6'b100001: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lh
127                     6'b100101: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lhu
128                     6'b001111: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lui
129                     6'b100011: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lw
130                     6'b110001: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// lwcl
131                     6'b001101: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// ori
132                     6'b101000: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// sb
133                     6'b001010: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// slti
134                     6'b001011: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// sltiu
135                     6'b101001: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// sh
136                     6'b101011: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 0; Imm = imm; end// sw
```

```
136                     6'b101011: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 0; Imm = imm; end// sw
137                     6'b111001: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// swcl
138                     6'b001110: begin RegA = rs; RegB = rt; RegDst = 0; RegWrite = 1; Imm = imm; end// xori
139
140             //J
141                     6'b000010: begin Target = target; RegDst = 0; RegWrite = 1; end// j
142                     6'b000011: begin Target = target; RegDst = 0; RegWrite = 1; end// jal
143
144                     default:
145                             begin
146                                     RegA = 5'b00000;
147                                     RegB = 5'b00000;
148                                     Imm = 16'b0000000000000000;
149                                     Target = 26'b00000000000000000000000000;
150                             end
151             endcase
152     end
153 endmodule
154
```

Testing

Since this is one of the most crucial parts of our project we wanted to make sure that we thoroughly tested it, and that each time an instruction was passed that it was being decoded correctly and passing back the correct values.

We are using the fibonacci machine code for testing this stage. We will be running the machine code produced by the MIPS Fibonacci code on our synthesized hardware. We will need to test the output of every instruction.

MIPS Fibonacci Sequence Code to be tested:

```
    .data
fibs: .word   0 : 12       # "array" of 12 words to contain fib values
size: .word  12            # size of "array"
    .text
    la   $t0, fibs       # load address of array
    la   $t5, size       # load address of size variable
    lw   $t5, 0($t5)     # load array size
    li   $t2, 1          # 1 is first and second Fib. number
    add.d $f0, $f2, $f4
    sw   $t2, 0($t0)     # F[0] = 1
    sw   $t2, 4($t0)     # F[1] = F[0] = 1
    addi $t1, $t5, -2    # Counter for loop, will execute (size-2) times
loop: lw   $t3, 0($t0)     # Get value from array F[n]
    lw   $t4, 4($t0)     # Get value from array F[n+1]
    add  $t2, $t3, $t4    # $t2 = F[n] + F[n+1]
    sw   $t2, 8($t0)     # Store F[n+2] = F[n] + F[n+1] in array
    addi $t0, $t0, 4     # increment address of Fib. number source
    addi $t1, $t1, -1    # decrement loop counter
    bgtz $t1, loop       # repeat if not finished yet.
    la   $a0, fibs       # first argument for print (array)
    add  $a1, $zero, $t5  # second argument for print (size)
    jal  print           # call print routine.
    li   $v0, 10         # system call for exit
    syscall              # we are out of here.

#########  routine to print the numbers on one line.
    .data
space:.asciiz  " "         # space to insert between numbers
head: .asciiz  "The Fibonacci numbers are:\n"
    .text
print:add  $t0, $zero, $a0  # starting address of array
    add  $t1, $zero, $a1  # initialize loop counter to array size
```

```
    la   $a0, head       # load address of print heading
    li   $v0, 4          # specify Print String service
    syscall              # print heading
out:  lw   $a0, 0($t0)     # load fibonacci number for syscall
    li   $v0, 1          # specify Print Integer service
    syscall              # print fibonacci number
    la   $a0, space      # load address of spacer for syscall
    li   $v0, 4          # specify Print String service
    syscall              # output string
    addi $t0, $t0, 4     # increment address
    addi $t1, $t1, -1    # decrement loop counter
    bgtz $t1, out        # repeat if not finished
    jr   $ra             # return
```

Below is the Machine Code that will be used to test Synthesized Hardware. Each line is the hexadecimal representation of the instruction being executed. The most significant 6 bits will be sent to the Control Unit as input.

These instruction Hexadecimals are the “FIBONACCI” data dump and sample segment.

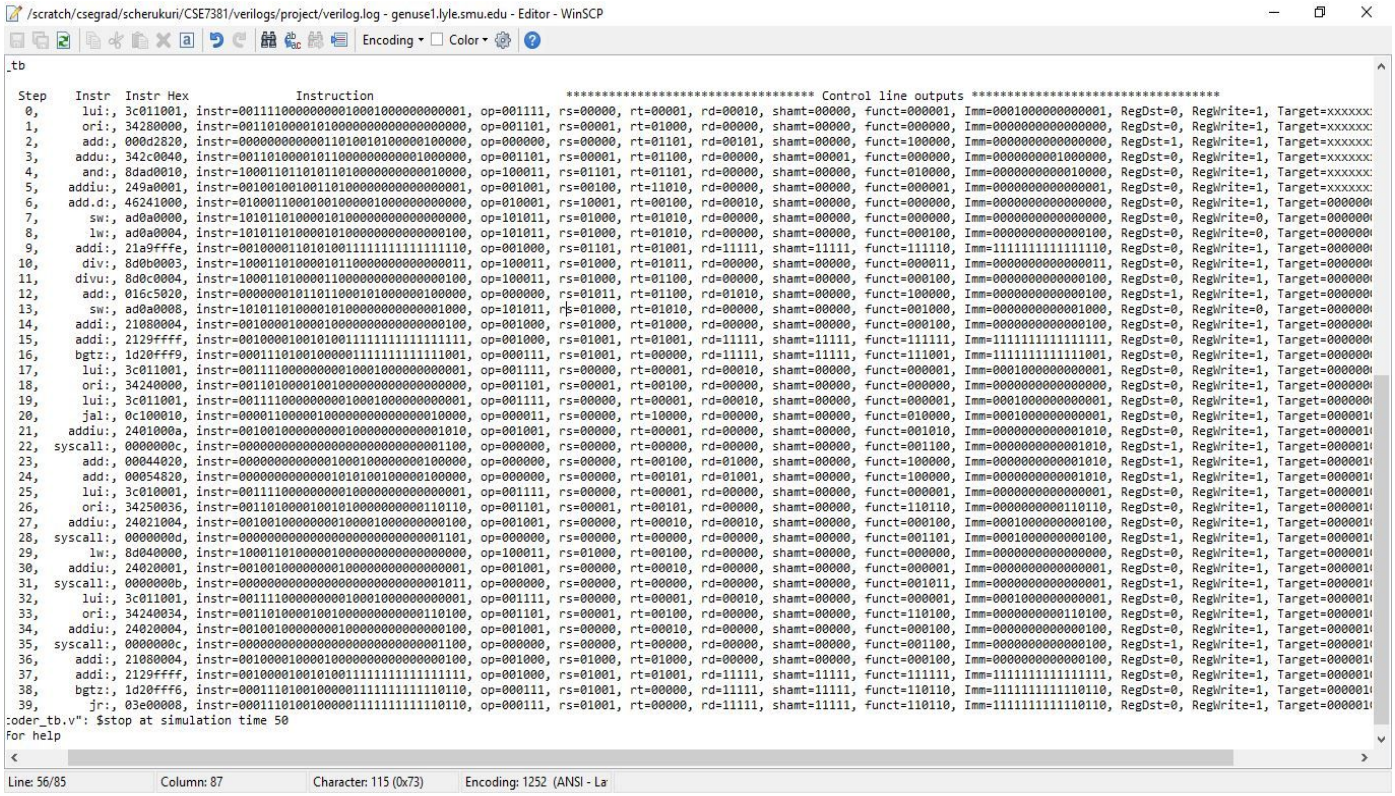
```
Step    Instr  Instr Hex
 0,      lui:, 3c011001, i
 1,      ori:, 34280000, i
 2,      add:, 000d2820, i
 3,     addu:, 342c0040, i
 4,      and:, 8dad0010, i
 5,    addiu:, 249a0001, i
 6,    add.d:, 46241000, i
 7,       sw:, ad0a0000, i
 8,       lw:, ad0a0004, i
 9,     addi:, 21a9fffe, i
10,      div:, 8d0b0003, i
11,     divu:, 8d0c0004, i
12,      add:, 016c5020, i
13,       sw:, ad0a0008, i
14,     addi:, 21080004, i
15,     addi:, 2129ffff, i
16,     bgtz:, 1d20fff9, i
17,      lui:, 3c011001, i
18,      ori:, 34240000, i
19,      lui:, 3c011001, i
20,      jal:, 0c100010, i
21,    addiu:, 2401000a, i
22,  syscall:, 0000000c, i
23,      add:, 00044020, i
24,      add:, 00054820, i
25,      lui:, 3c010001, i
26,      ori:, 34250036, i
27,    addiu:, 24021004, i
28,  syscall:, 0000000d, i
```

Testing Code

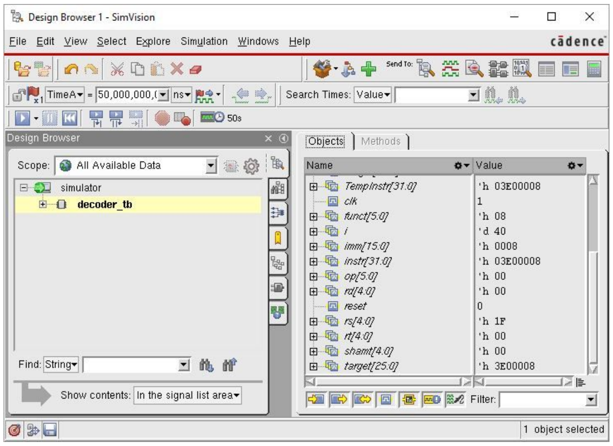
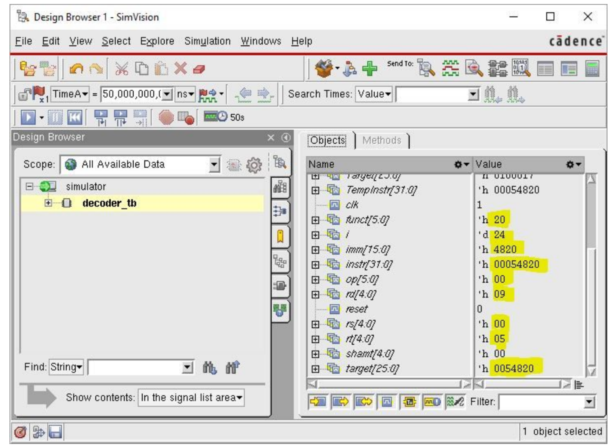
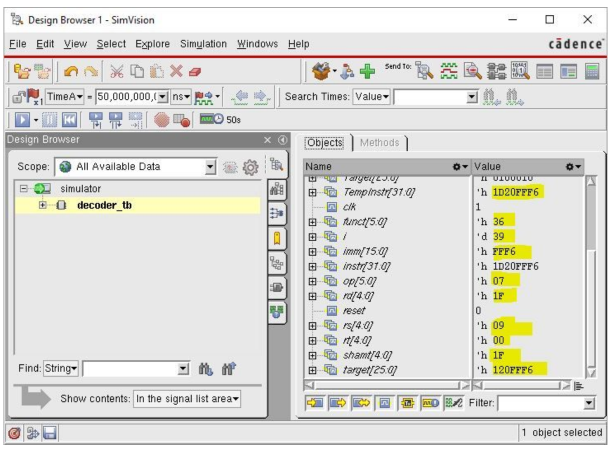
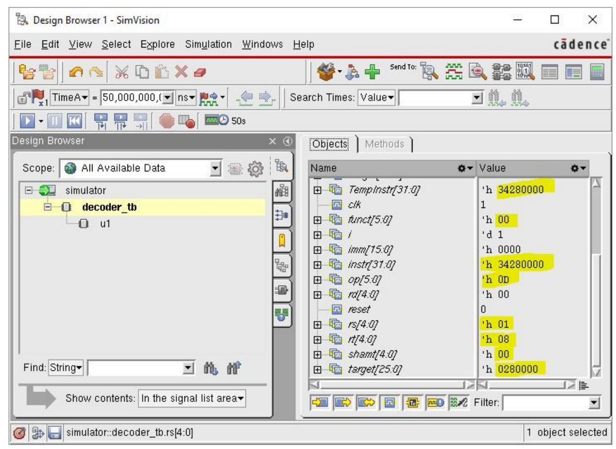
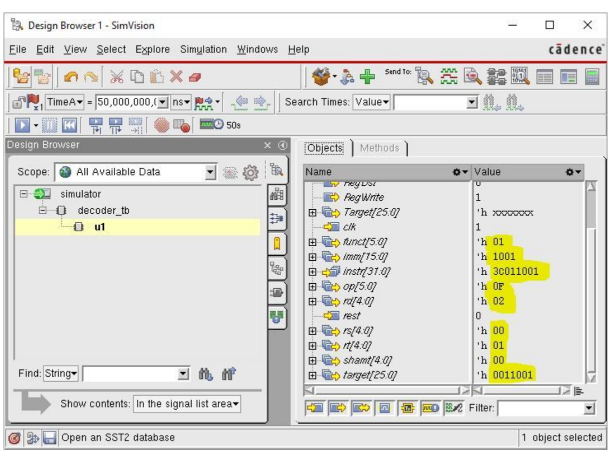
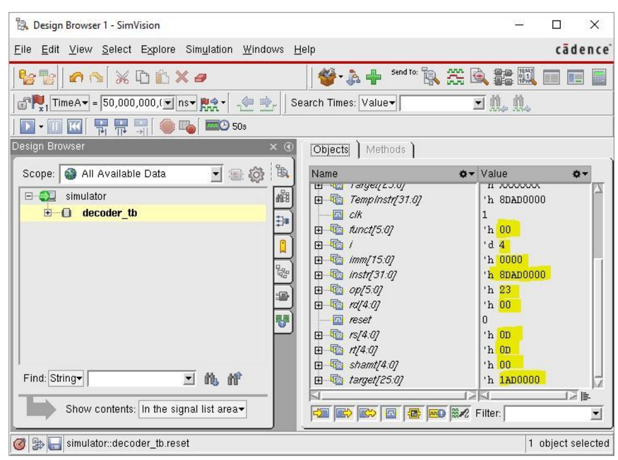
```
module decoder_tb;

    reg clk;
    reg reset;
    reg [31:0] instr;

    wire [5:0] op;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [4:0] shamt;
    wire [5:0] funct;
    wire [15:0] imm;
    wire [25:0] target;
    wire [25:0] Target;
    wire [15:0] Imm;

    wire RegDst, RegWrite;
    wire [4:0] RegA, RegB;

    reg [31:0] TempInstr;
    reg [31:0] InstrArray [0:39];
    reg [8*7:0] NameArray [0:39];
    integer i;

    decoder u1(instr, clk, reset, op, rs, rt, rd, shamt, funct, imm, target, RegA, RegB, Imm, RegDst, RegWrite, Target);

    initial
    begin
    clk = 0;
    reset = 0;
    end

    always
    #5  clk =  ! clk;

    initial

     begin
     $display("         Step    Instr  Instr Hex              Instruction

     InstrArray[0] = 32'h 3c011001; NameArray[0] = "lui";
     InstrArray[1] = 32'h 34280000; NameArray[1] = "ori";
     InstrArray[2] = 32'h 000d2820; NameArray[2] = "add";
     InstrArray[3] = 32'h 342c0040; NameArray[3] = "addu";
     InstrArray[4] = 32'h 8dad0010; NameArray[4] = "and";
     InstrArray[5] = 32'h 249a0001; NameArray[5] = "addiu";
     InstrArray[6] = 32'h 46241000; NameArray[6] = "add.d";
     InstrArray[7] = 32'h ad0a0000; NameArray[7] = "sw";
     InstrArray[8] = 32'h ad0a0004; NameArray[8] = "lw";
     InstrArray[9] = 32'h 21a9fffe; NameArray[9] = "addi";
     InstrArray[10] = 32'h 8d0b0003; NameArray[10] = "div";
     InstrArray[11] = 32'h 8d0c0004; NameArray[11] = "divu";
     InstrArray[12] = 32'h 016c5020; NameArray[12] = "add";
     InstrArray[13] = 32'h ad0a0008; NameArray[13] = "sw";
     InstrArray[14] = 32'h 21080004; NameArray[14] = "addi";
     InstrArray[15] = 32'h 2129ffff; NameArray[15] = "addi";
     InstrArray[16] = 32'h 1d20fff9; NameArray[16] = "bgtz";
     InstrArray[17] = 32'h 3c011001; NameArray[17] = "lui";
     InstrArray[18] = 32'h 34240000; NameArray[18] = "ori";
     InstrArray[19] = 32'h 3c011001; NameArray[19] = "lui";
     InstrArray[20] = 32'h 0c100010; NameArray[20] = "jal";
     InstrArray[21] = 32'h 2401000a; NameArray[21] = "addiu";
     InstrArray[22] = 32'h 0000000c; NameArray[22] = "syscall";
     InstrArray[23] = 32'h 00044020; NameArray[23] = "add";
     InstrArray[24] = 32'h 00054820; NameArray[24] = "add";
     InstrArray[25] = 32'h 3c010001; NameArray[25] = "lui";
     InstrArray[26] = 32'h 34250036; NameArray[26] = "ori";
     InstrArray[27] = 32'h 24021004; NameArray[27] = "addiu";
     InstrArray[28] = 32'h 0000000d; NameArray[28] = "syscall";
```

```
70      InstrArray[29] = 32'h 8d040000; NameArray[29] = "lw";
71      InstrArray[30] = 32'h 24020001; NameArray[30] = "addiu";
72      InstrArray[31] = 32'h 0000000b; NameArray[31] = "syscall";
73      InstrArray[32] = 32'h 3c011001; NameArray[32] = "lui";
74      InstrArray[33] = 32'h 34240034; NameArray[33] = "ori";
75      InstrArray[34] = 32'h 24020004; NameArray[34] = "addiu";
76      InstrArray[35] = 32'h 0000000c; NameArray[35] = "syscall";
77      InstrArray[36] = 32'h 21080004; NameArray[36] = "addi";
78      InstrArray[37] = 32'h 2129ffff; NameArray[37] = "addi";
79      InstrArray[38] = 32'h 1d20fff6; NameArray[38] = "bgtz";
80      InstrArray[39] = 32'h 03e00008; NameArray[39] = "jr";
81      end
82
83      initial
84      begin
85      for(i=0; i<39; i=i+1) begin
86      TempInstr = InstrArray[i];
87      instr = TempInstr;
88      #1;
89      end
90      end
91
92      // initial $monitor("%d, %s:, %h, instr=%b, clk=%b, reset=%b, op=%b, rs=%b, rt=%b, rd=%b, shamt=%b, funct=%b,
93      // i, NameArray[i], InstrArray[i], instr, clk, reset, op, rs, rt, rd, shamt, funct, imm, target, RegA, RegB, I
94
95      initial $monitor("%d, %s:, %h, instr=%b, op=%b, rs=%b, rt=%b, rd=%b, shamt=%b, funct=%b, Imm=%b, RegDst=%b,
96                        RegWrite=%b, Target=%b",
97      i, NameArray[i], InstrArray[i], instr, op, rs, rt, rd, shamt, funct, Imm, RegDst, RegWrite, Target);
98
99      initial #50 $stop;
100
101   endmodule // decoder_tb
102
```

Test Output: “verilog +gui decoder.v decoder_tb.v &”



Finally, the testbench results printout to the console window so the final validation processes can be completed. In the below console window screen shots, we show the necessary output for every instructions and types (R-type, I-type, and J-type).

4. MUX input to Write register on Registers

Original Design

According to 3 types of MIPS instructions, R-type and I-type instructions require *register file* write data from MEM stage into Write register. However, these two type instructions have different destination location bits. For instance,

R-type destination register(Rd) locates between sixteenth and eleventh bits, Rd(15-11)

I-type destination register(Rt) locates between twenty-first and seventeenth bits, Rt(20-16)

Therefore, CPU set *RegDst* as a control signal to determine which register number will be passed by using *Write register MUX*. When the instruction is I-type, *RegDst* set as 0, MUX will transport 20-16 bits into *Write register*. On the contract, when the instruction is R-type, *RegDst* set as 1, MUX will transport 15-11 bits into *Write register.*

Implementation

Our implementation for the multiplexer has 4 inputs. The first two inputs, labelled input1 and input2 respectively, are used to hold the two different destination registers, and as such are both 5 bits long. Input1 holds the destination register if the instruction is an I-type while input2 holds the destination register for R-type instructions. The third input is the select input. It is a binary output, so it can be either 0 or 1. We use this output to determine whether to use input1 as the address when select = 0, or to use input2 when select =1. The last input to the module is actually called out, and this is used to pass back the selected input.

Code Listing

```
2   module multiplexer(
3     input1,
4     input2,
5     select,
6     out
7     );
8
9     input [4:0] input1, input2;
10    input select;
11    output [4:0] out;
12
13    always @ ( input1 or input2 or select ) begin
14        out = select ? input2 or input1;
15    end
16
17  endmodule
18
```

5. Sign Extend

Original Design

Sign extend is used to extend a 16 bit operand given into by an instruction to 32 bits while preserving the sign of the original number. To do this, you look at the leftmost bit, and you extend that bit another 16 bits.

Sign Extending 10:

Original: 0000 0000 0000 1010

New: 0000 0000 0000 0000 0000 0000 0000 1010

Sign Extending -15:

Original: 1111 1111 1111 0001

New 1111 1111 1111 1111 1111 1111 1111 0001

To test that our implementation is correct, we will first do basic test where we pass in 16-bit numbers, and check that the corresponding output is as it should be. The second stage of testing will be to connect it to the instruction decoder, and once again, make sure that all outputs are consistent with the numbers that should be coming out.

Implementation

The module is passed two variables, one input, in, that represents the number to be extended, and one output, out, that represents the number once it has been extended to 32 bits. The function determines whether bit 15 of variable in is a 1 or a zero, and then it extends that out so that the number will be 32 bits.

Code Listing

```
1
2  module sign_extender(in, out);
3
4    input [15:0] in;
5    output [31:0] out;
6
7    assign out = {{16{in[15]}}, in[15:0]};
8
9  endmodule
10
```

Testing

To test that our implementation is correct, we pass in 16-bit numbers, and check that the corresponding output is as it should be.

Testing Code: “verilog +gui sign_extend.v sign_extender_tb.v &”

```
1
2
3  module test_sign_extend;
4
5    reg [15:0] in;
6    wire [31:0] out;
7
8    sign_extender DUT(in, out);
9
10   initial begin
11     #1 in = 16'h 7fff;
12     #1 in = 16'h 8fff;
13     #1 in = 16'h 7001;
14     #1 in = 16'h 7121;
15     #1 in = 16'h 800f;
16   end
17
18   initial $monitor($time, " in=%h in=%b out=%b",
19     in, in, out);
20   initial #20 $stop;
21
22 endmodule
23
```

Test Output

```
Compiling source file "sign_extend.v"
Compiling source file "sign_extender_tb.v"
Highest level modules:
test_sign_extend

simvision(64): 15.20-s036: (c) Copyright 1995-2017 Cadence Design Systems, Inc.
waiting for SimVision to connect...
waiting for SimVision to connect...

------------------------------------
Relinquished control to SimVision.....
                   0 in=xxxx in=xxxxxxxxxxxxxxxx out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                   1 in=7fff in=0111111111111111 out=00000000000000000111111111111111
                   2 in=8fff in=1000111111111111 out=11111111111111111000111111111111
                   3 in=7001 in=0111000000000001 out=00000000000000000111000000000001
                   4 in=7121 in=0111000100100001 out=00000000000000000111000100100001
                   5 in=800f in=1000000000001111 out=11111111111111111000000000001111
```

References:

[1] https://en.wikibooks.org/wiki/MIPS_Assembly/Register_File
[2] https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec08.pdf
[3] http://www-inst.eecs.berkeley.edu/~cs150/sp12/lab3/