Lab 02:
Assembly Programming Setup & HW 1

Hunjun Lee
hunjunlee@hanyang.ac.kr

What we will do …

- We will eventually implement the CPU for MIPS architecture
- For debugging purpose, you will need to implement a testbench to check if your CPU is working properly
- For that, you need to know how to do an assembly programming
  - Refer to the following link for the detailed MIPS ISA
    - https://opencores.org/projects/plasma/opcodes

Target MIPS Assembly - 1

Assembly	Action	Opcode bitfields					
ADDU rd, rs, rt	rd = rs + rt	000000	rs	rt	rd	00000	100001
ADDIU rt, rs, imm	rt = rs + sign(imm)	001001	rs	rt	imm		
AND rd, rs, rt	rd = rs & rt	000000	rs	rt	rd	00000	100100
ANDI rt, rs, imm	rt = rs & zero(imm)	001111	rs	rt	imm		
LUI rt,imm	rt = imm << 16	000000	rs	rt	imm		
NOR rd,rs,rt	rd = ~(rs | rt)	000000	rs	rt	rd	00000	100111
OR rd,rs,rt	rd = rs | rt	000000	rs	rt	rd	00000	100101
ORI rt,rs,imm	rt = rs | zero(imm)	001101	rs	rt	imm		
SLT rd,rs,rt	rd = rs < rt	000000	rs	rt	rd	00000	101010
SLTI rt,rs,imm	rt = rs < sign(imm)	001010	rs	rt	imm		
SLTIU rt,rs,imm	rt = rs < sign(imm)	001011	rs	rt	imm		
SLTU rd,rs,rt	rd = rs < rt	000000	rs	rt	rd	00000	101011
SUBU rd,rs,rt	rd = rs - rt	000000	rs	rt	rd	00000	100011
XOR rd,rs,rt	rd = rs ^ rt	000000	rs	rt	rd	00000	101010
XORI rt,rs,imm	rt = rs ^ zero(imm)	001010	rs	rt	imm		

Target MIPS Assembly - 2

Assembly	Action	Opcode bitfields					
SLL rd, rt, sa	rd = rs << sa	000000	rs	rt	rd	sa	000000
SRA rd, rt, sa	rd = rt >> sa (arithmetic)	000000	rs	rt	rd	sa	000011
SRL rd, rt, sa	rd = rt >> sa (logical)	000000	rs	rt	rd	sa	000010
BEQ rs, rt, offset	if(rs == rt) pc += sign(offset) << 2	000100	rs	rt	offset		
BNE rs, rt, offset	if(rs != rt) pc += sign(offset) << 2	000101	rs	rt	offset		
J target	pc=pc_upper|(target << 2)	000010	target				
JAL target	r31 = pc; pc=pc_upper|(target << 2)	000011	target				
JR rs	pc = rs	000000	rs	000000000000000			001000
LW rt, offset(rs)	rt = MEM[sign(offset) + rs]	100011	rs	rt	offset		
SW rt, offset(rs)	MEM[sign(offset) + rs] = rt	101011	rs	rt	offset		

You'll be implementing a CPU that supports these ISAs
Use these for your assembly programming!

Some minor issues

- `add vs. addu / sub vs. subu`
  - There is no fundamental difference between signed and unsigned operations (for add and subtraction)
  - In the following projects, we will only use unsigned add and subtract operations

- signed vs. zero extension
  - Some operations require zero extension (e.g., logical immediate operations)

MIPS Assembly Setup

- You need to use the MIPS emulator
  - download from the link below …
    - https://dpetersanderson.github.io/download.html

- Program an assembly (.asm file) and you can emulate how each instruction modifies the architectural state
  - Register file values
  - Memory data
  - Program counter value

- Again, this is to make a testbench for your MIPS CPU implementation in Verilog (later projects)
  - Not mandatory … but it will help you ‘a lot’

Programming MIPS Assembly

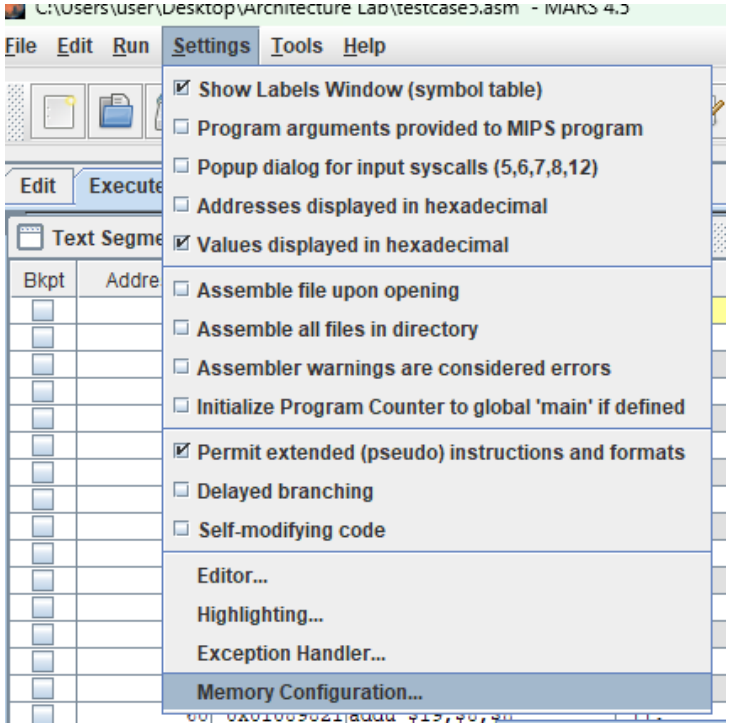
- You can program using the MIPS assembly in the previous slides

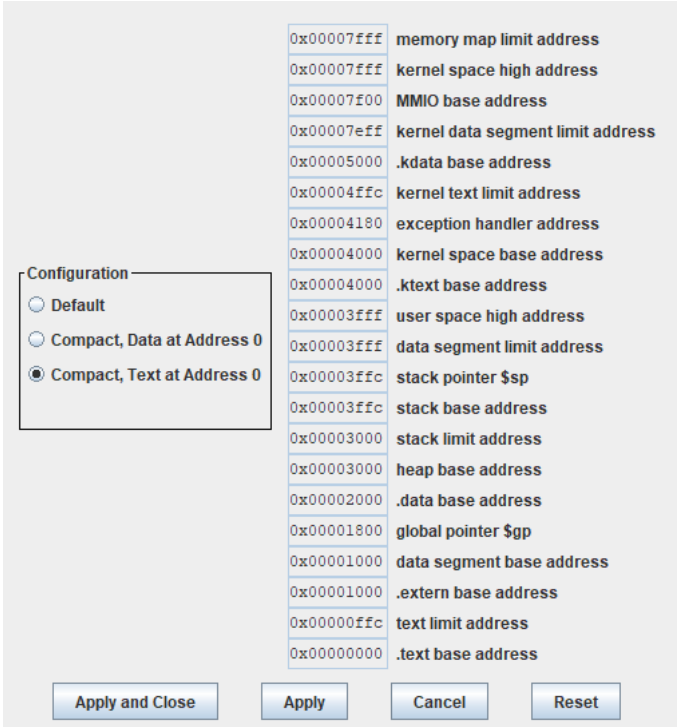
- By the way … Mars supports some high-level assemblies that are decomposed into multiple low-level assemblies
  - Examples …
  - `li $t0, 1000000`
    - `lui $at, 0x0000000f`
    - `ori $t0, $at, 0x00004240`

- Practice … with various programs

Memory Setup

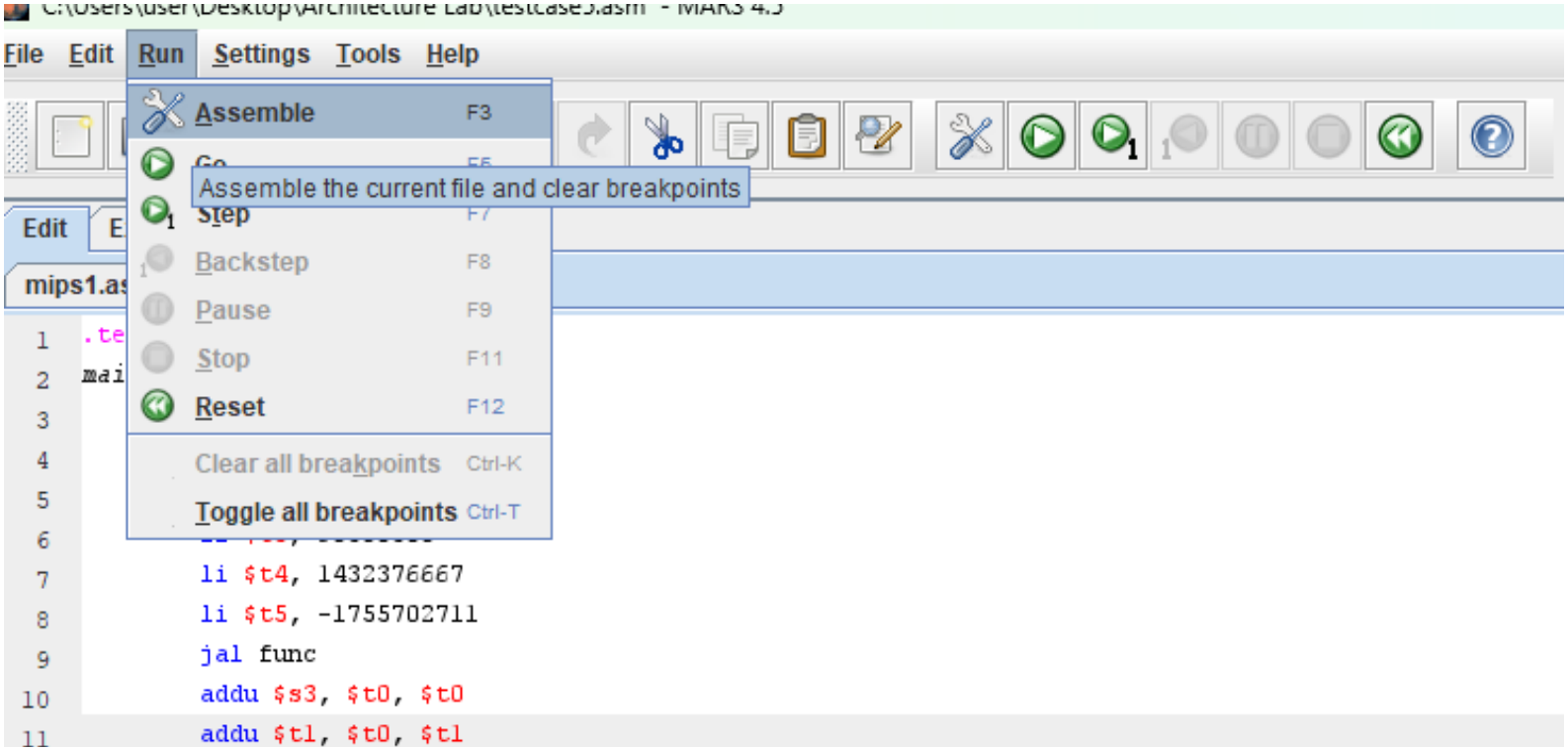
- We'll be designing a "small" memory setup that does not fully use the 32-bit address space
- Use the compact memory format …
  - Our CPU implementation will assume the exactly same memory format





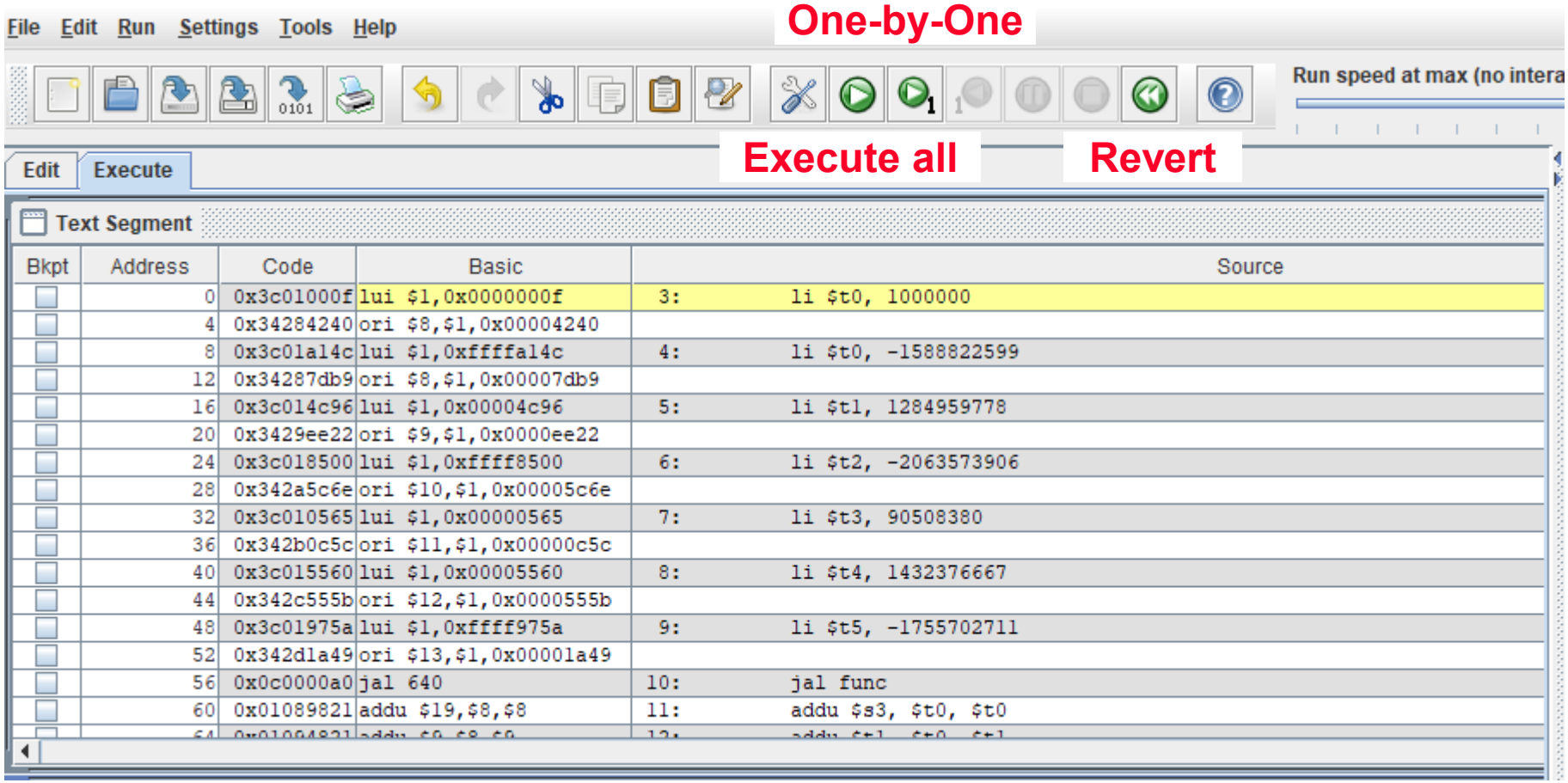
Compile

- After programming, you can assemble the code which converts the MIPS assembly program into the machine code



Execution

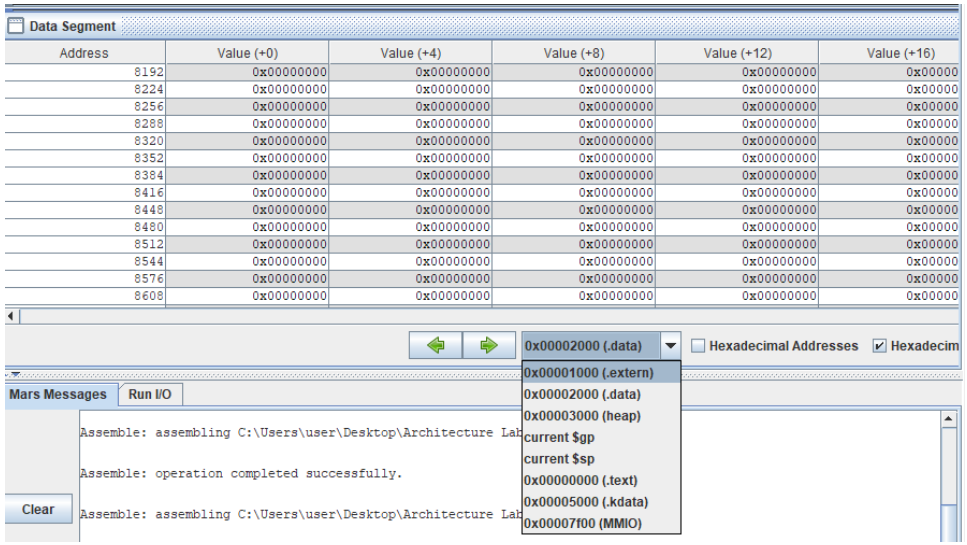
- You can execute each instruction one by one …



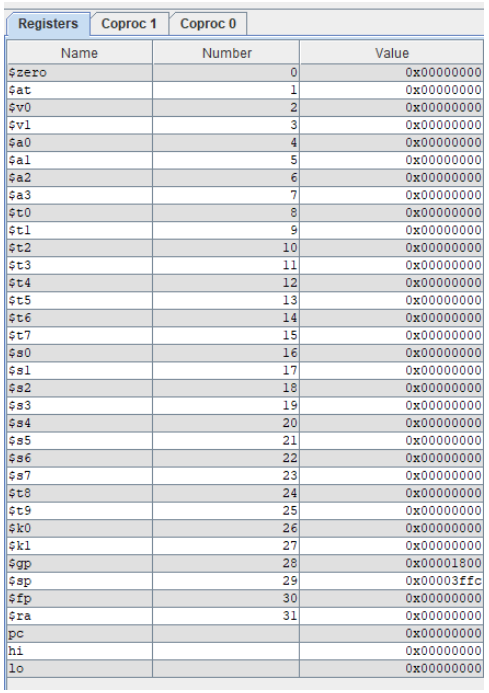
Bkpt	Address	Code	Basic	Source
	0	0x3c01000f	lui $1,0x0000000f	3: li $t0, 1000000
	4	0x34284240	ori $8,$1,0x00004240	
	8	0x3c01a14c	lui $1,0xffffa14c	4: li $t0, -1588822599
	12	0x34287db9	ori $8,$1,0x00007db9	
	16	0x3c014c96	lui $1,0x00004c96	5: li $t1, 1284959778
	20	0x3429ee22	ori $9,$1,0x0000ee22	
	24	0x3c018500	lui $1,0xffff8500	6: li $t2, -2063573906
	28	0x342a5c6e	ori $10,$1,0x00005c6e	
	32	0x3c010565	lui $1,0x00000565	7: li $t3, 90508380
	36	0x342b0c5c	ori $11,$1,0x00000c5c	
	40	0x3c015560	lui $1,0x00005560	8: li $t4, 1432376667
	44	0x342c555b	ori $12,$1,0x0000555b	
	48	0x3c01975a	lui $1,0xffff975a	9: li $t5, -1755702711
	52	0x342d1a49	ori $13,$1,0x00001a49	
	56	0x0c0000a0	jal 640	10: jal func
	60	0x01089821	addu $19,$8,$8	11: addu $s3, $t0, $t0

Architectural States

- You can see the architectural states as you execute each instruction in sequence



Memory Data (at difference region …)



Register file data

Use these to debug your cpp and Verilog implementation in the later projects!

Now … about your project!

Basic ALU Implementation

- You will design an ALU for your MIPS CPU
  - The ALU receives two 32-bit operands
    - `operand1`
    - `operand2`
  - A 5-bit shift amount (for shift operations)
    - `shamt`
  - A 4-bit control signal
    - `aluop`
  - A 32-bit output result
    - `alu_result`

- You will implement the ALU in both cpp and Verilog

Overview

- To operate as a MIPS ALU, it needs to support the following operations

Aluop	Operations	Action
ALU_ADDU	Add (w/o overflow)	alu_result = operand1 + operand2
ALU_AND	AND	alu_result = operand1 and operand2
ALU_NOR	NOR	alu_result = operand1 nor operand2
ALU_OR	OR	alu_result = operand1 or operand2
ALU_SLL	Shift left logical	alu_result = operand2 << shamt
ALU_SRA	Shift right arithmetic	alu_result = operand2 >>> shamt
ALU_SRL	Shift right logical	alu_result = operand2 >> shamt
ALU_SUBU	Sub (w/o overflow)	alu_result = operand1 - operand2
ALU_XOR	XOR	alu_result = operand1 xor operand2
ALU_SLT	Set less than	alu_result = operand1 < operand2
ALU_SLTU	Set less than unsigned	alu_result = unsigned(operand1) < unsigned(operand2)
ALU_EQ	Equal	alu_result = operand1 == operand2
ALU_NEQ	Not equal	alu_result = operand1 != operand2
ALU_LUI	Set upper bits	alu_result = operand2 << 16

Overview …

- You will be given two different files
  - `global.h` for the high-level simulator
  - `GLOBAL.v` for the Verilog simulator

```
enum ALUOp {
    ALU_ADD = 0,
    ALU_AND = 1,
    ALU_NOR = 2,
    ALU_OR = 3,
    ALU_SLL = 4,
    ALU_SRA = 5,
    ALU_SRL = 6,
    ALU_SUB = 7,
    ALU_XOR = 8,
    ALU_SLT = 9,
    ALU_EQ = 10,
    ALU_NEQ = 11,
    ALU_LUI = 12
};
```

```
`define     ALU_ADDU        4'b0000
`define     ALU_AND         4'b0001
`define     ALU_NOR         4'b0010
`define     ALU_OR          4'b0011
`define     ALU_SLL         4'b0100
`define     ALU_SRA         4'b0101
`define     ALU_SRL         4'b0110
`define     ALU_SUBU        4'b0111
`define     ALU_XOR         4'b1000
`define     ALU_SLT         4'b1001
`define     ALU_SLTU        4'b1010
`define     ALU_EQ          4'b1011
`define     ALU_NEQ         4'b1100
`define     ALU_LUI         4'b1101
```

Debugging CPP

- I made 100 reference tests to check the functionality of the CPP

```
// I'm providing 100 reference tests!
ifstream fin_op1_ref;
ifstream fin_op2_ref;
ifstream fin_shamt_ref;
ifstream fin_funct_ref;
ifstream fin_result_ref;
fin_op1_ref.open("operand1.ref");
fin_op2_ref.open("operand2.ref");
fin_shamt_ref.open("shamt.ref");
fin_funct_ref.open("funct.ref");
fin_result_ref.open("alu_result.ref");

int PASSED = 0;
int FAILED = 0;
for (int cycle = 0; cycle < 100; cycle++) {
    fin_op1_ref >> operand1;
    fin_op2_ref >> operand2;
    fin_shamt_ref >> shamt;
    fin_funct_ref >> aluop;
    fin_result_ref >> alu_result_ref;

    alu.compute(operand1, operand2, shamt, aluop, &alu_result);

    if (alu_result == alu_result_ref)
        PASSED += 1;
    else
        FAILED += 1;

}
cout << "PASSED: " << PASSED << ", FAILED: " << FAILED << endl;
```

Debugging Verilog - 1

- The CPP simulator generates reference data to debug the Verilog modules!

```
// Make a custom testbench using our cpp simulator!
ofstream fout_op1;
ofstream fout_op2;
ofstream fout_shamt;
ofstream fout_funct;
ofstream fout_result;
fout_op1.open("operand1.mem");
fout_op2.open("operand2.mem");
fout_shamt.open("shamt.mem");
fout_funct.open("funct.mem");
fout_result.open("alu_result.mem");


// This is the provided example, but you can try out different cases (if you want)
for (int cycle = 0; cycle < 1000; cycle++) {
    operand1 = ((rand() << 1) + rand());
    operand2 = ((rand() << 1) + rand());
    shamt = rand() % 32;
    aluop = rand() % 14;

    alu.compute(operand1, operand2, shamt, aluop, &alu_result);

    // Write down the file! (to use as testbench!)
    fout_op1 << std::setw(8) << std::setfill('0') << std::hex << operand1 << endl;
    fout_op2 << std::setw(8) << std::setfill('0') << std::hex << operand2 << endl;
    fout_shamt << std::bitset<5>(shamt) <<endl;
    fout_funct << std::bitset<4>(aluop) <<endl;
    fout_result << std::setw(8) << std::setfill('0') << std::hex << alu_result << endl;
}
```

Debugging Verilog - 2

- Verilog testbench utilizes the generate .mem files to debug your hardware module …

```
task Test;
    input [31:0] operand1_in;
    input [31:0] operand2_in;
    input [4:0] shamt_in;
    input [3:0] funct_in;
    input [31:0] alu_result_in;
begin
    operand1 = operand1_in;
    operand2 = operand2_in;
    shamt = shamt_in;
    funct = funct_in;
    $display("TEST CTRL Sig: %h :", funct);
    #1;
    if (alu_result == alu_result_in) begin
        $display("PASSED");
        PASSED = PASSED + 1;
    end
    else begin
        $display("FAILED");
        $display("funct: %d, operand1 = %d, operand2 = %d, shamt = %d, result = %h (Ans : %h)",
                    funct, operand1, operand2, shamt, alu_result, alu_result_in);
        FAILED = FAILED + 1;
    end
end
endtask

// Load the data from the memory
initial begin
    $readmemh("operand1.mem", operand1_mem);
    $readmemh("operand2.mem", operand2_mem);
    $readmemb("shamt.mem", shamt_mem);
    $readmemb("funct.mem", funct_mem);
    $readmemh("alu_result.mem", alu_result_mem);
end
```

Submission

- ALU implementation modify ALU.cpp and ALU.v
- Submission (Zip all the files)
  - For a two-people team: lab1_student_id1_student_id2.zip
  - For a one-person team: lab1_student_id.zip
    - You must follow the format (-10% for wrong file format)
    - Example:
      - lab1_2020102030.zip
      - lab1_2020102030_2022103040.zip
  - The zip file should contain:
    - ALU.cpp, ALU.v
    - lab1_report.pdf

Submission

- You need to write 4-page report for each submission
  - Explain the overall structure and how you implemented each program
  - Write down the difficulties if you had one
  - Draw the hardware modules if needed …

- Due: Fri. March 28st
  - 1 week delay: -20%
  - 2 week delay: -50%
  - Further delay: 0 point