Sam Nguyen

Istiak Rahman

Neal Prajapati

ECE-355

2/18/2024

Project 2 Summary

For Section 5, we were tasked with storing the array arr[3] = {19088743, 2882400001, 169552957} into a size-of24-bytes data memory. Since RISCV is little endian the least-significant byte should go to the least address. Another thing they will help out writing this down would to convert the long ints into hex. Hex makes the numbers look a little messy. For problem 1a, one integer takes up 8 bytes. This makes sense because the integers in these are arrays or long int, which take up 8 bytes in the memory. If we were dealing with regular integers, they would take up 4 bytes. For problem 1b, since we mentioned that converting the long integers to hex would make the calculations easier to work with, we converted the long integers to long ints:

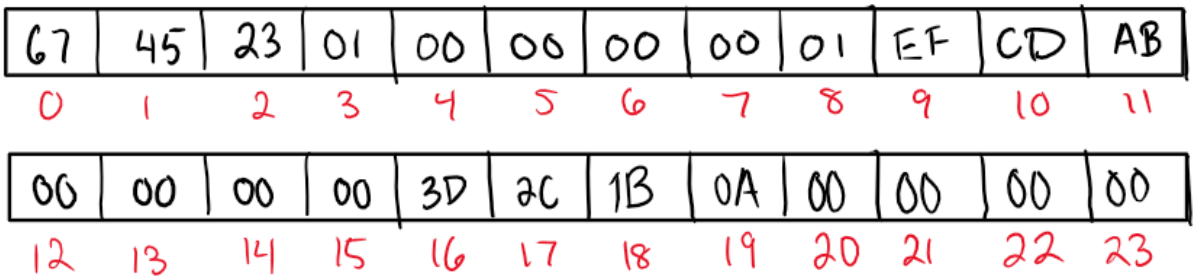
arr[0] = 19088743 = 0x0000000001234567

arr[1] = 2882400001 = 0x00000000ABCDEF01

arr[2] = 169552957 = 0x000000000A1B2C3D

With this is mind, the representation in memory is shown below:

RISC-V: Little Endian Representation



For problem 2, we need to execute the commands ld x8, 0(x23), ld x9, 16(x23), and add x10, x8, x9. To execute these instructions, we would need to set register 8 and 9 equal certain parts in register 10. The calculation is shown below:

x10 = x8 + x9 Base Address: x23 = 0

x8 = 0(x23) = 01 23 45 67

\+ x9 = 16(x23) = 0A 1B 2C 3D

x10 = 0B 3E 71 A4

For Section 6, we are tasked with simulating the behavior of a single-cycle RISC-V CPU. To simulate this behavior, we needed to set up the instruction memory and data memory. This is accomplished by the "initCore()" function in Core.c. "initCore()" accepts the instruction memory as a parameter that we coded in project 1, and then mallocs a core so that we can now initialize variables we will need such as the clock, pointer counter, instruction memory, and the tick. "initCore()" also sets all the values in its register filers and data memory to 0 so that no data is held there when we start to execute the instructions. We also have to set some register values and data memory after this so that we get our desired results after running some of the instructions.

After core is initialized with the configurations that were needed, we enter a while loop which will loop the function "tickFunc()" until we are at the end at the instructions thanks to the pointer counter. "tickFunc()" is the main function that will be executing the instructions one by one. The first line that "tickFunc()" is fetching the instruction that the PC is pointing to as an unsigned int. It then initializes all the variables needed in an instruction such as the opcode, rs1, rs2, rd, funct3, funct7, and the immediate. We then call the "instruct_split()" function to split up the instruction to its respective parts. The opcode is split first to determine which type of instruction we are dealing with, and from there, we split up the rest of the parts based on what each instruction needs. The special case in this code however is SB and I type since the immediate has to be signed. To solve this special case, we can use the "extractImmediate()" function which extracts the immediate out of the sb and I type instructions.

After initializing the parts of the instructions, we initialize what needs to be active for the Control based on the opcode. This is accomplished by the "ControlUnit()" function. This function receives the opcode, and a pointer pointing to the control signals. Based on what opcode is, this function will "turn on" the signals that would need to be active for that instruction. After "ControlUnit()" is called, we check if the signal ALUSrc is turned on by using a MUX. If it was on, we use the immediate, but if it is not, we use rs2. We then determine which operation the instruction is using by utilizing the "ALUControlUnit()" function. The function receives the signal ALOp, funct7, and funct3. Using these three variables, this function determines which operation the ALU needs to compute, and then returns a number corresponding to sed operation.

From there, we check again which type of instruction we are dealing with, and from there, we perform the respective operation by calling the "ALU()" function. The ALU function is simple in that fact that it just stores the result of the operation in ALU result, or if the result is zero, then it makes the signal zero equal to 1. The only exceptions to these instructions don't do a typical operation, but rather stores and loads variables to data memory or registers. These operations are accomplished accessing data memory or the register files, shifting that destination by the immediate, and then loading or storing the value that was asked to be stored there.

The last bits of lines that "tickFunc()" has is to update PC and clock. It then checks if we are at the last instruction. If we are at the last instruction, then we return false and while loop stops. If we are not, then we return true and then go to the next instruction. The final memory and register values are shown below:

```
Loading trace file: ./cpu_traces/project_two
INSTRU: 26543411
Opcode: 51
rd: 10, rs1: 10, funt3: 0, rs2: 25, funct7: 0
ALUOP:2, F7:0, F3:0
OP: 2
R-Type Memory Address: 0xa
10:8

INSTRU: 1085539379
Opcode: 51
rd: 8, rs1: 8, funt3: 0, rs2: 11, funct7: 16
ALUOP:2, F7:16, F3:0
OP: 6
R-Type Memory Address: 0x8
8:0

INSTRU: 341123
Opcode: 3
rd: 9, rs1: 10, funt3: 3, imm: 0
ALUOP:0, F7:0, F3:3
OP: 2
Before Change:0x558e5e557932
x9 = data_mem[8]: 0x558e5e557932
9:108

INSTRU: 1772307
Opcode: 19
rd: 22, rs1: 22, funt3: 0, imm: 1
ALUOP:0, F7:0, F3:0
OP: 2
I-Type Memory Address: 0x16
22:3

INSTRU: 3872147
Opcode: 19
rd: 11, rs1: 22, funt3: 1, imm: 3
ALUOP:0, F7:0, F3:1
OP: 61
I-Type Memory Address: 0xb
11:24

INSTRU: -8120093
Opcode: 99
rs1: 8, funt3: 1, rs2: 24, imm: 4294967280
ALUOP:1, F7:0, F3:1
OP: 20
0 vs 0
Simulation is finished.
```

Final Value & Memory Location

Registers	X8	X9	X10	X11	X22	X24
Final Value	0	108	8	24	3	0
Mem Address	0x8	0x558e5e557932	0xa	0xb	0x16	0x18