**IT3280E Computer Architecture Lab**

**Nguyễn Đức Anh**

**MSSV: 20235890**

**Lab 2. Instruction Set, Basic Instructions, Compiler Directives**

**Assignment 1**

Source code:

A number and equation written on a white background

Description automatically generated with medium confidence

We assemble the code, The PC value begins from the address of the first code line according to the text segment:

A screenshot of a computer

Description automatically generated

After we run next step, the value we add immediate to s0 from the code is 0x512 has assign to the value of the s0 at the registers windows and the PC value move to 0x00400004, that is the address of second line of code it’s ready to do next step:

A screenshot of a computer

Description automatically generated

We run the second line of code and the value changes to 0 because we do addition s0 = x0 + 0 with x0 = 0 so the value of s0 return to 0 and the pc value changes to 0x00400008:

A screenshot of a computer

Description automatically generated

Because the program does not have the third line so after we run next step the program end without any errors.

**Assignment 2**

Source code:

A screenshot of a computer

Description automatically generated

According to Assignment 1, the addi (addition immediate) is the I-type instruction so it’s only used to show the 12bits integer so we can’t use addi to assign the 32bits value.



Instead of that, we use lui (load upper immediate), it’s the u-type instruction allows us to assign the 20bits value as most significant bits to of 32bits value like the comment on the source code image and we use addi to add 12bits as least significant bits.

In this assignment, we want to add 0x20232024 is the 32bits value to s0. To do this, like the explanation, we do lui (load upper immediate) to load 0x20232 20bits most significant bits and use addi (addition immediate) to load 0x024 12bits least significant bits. The result we collect is 0x20232024 32bits value on s0 at end.

A screenshot of a computer

Description automatically generated

After we run the line 1, the s0 value is 0x20232000 and the PC value is 0x00400004 and its ready for next step.

A screenshot of a computer

Description automatically generated

After run line 2, the s0 value is completely loaded to the value 0x20232024 as explanation on the first and the program end when we run last step.

**Assignment 3**

Source code:

A screenshot of a white board with numbers and letters

Description automatically generated

In this assignment, we use li (load immediate) is the extended instruction, it’s not included in basic instructions of RISC-V but it’s popular on using by assemblers or programmers so whenever we use this, it complied to 1 or more real instructions.

A screenshot of a computer

Description automatically generated

As we can see, on the source code, we want to li (load immediate) the 32bits value 0x20232024 to s0 so it complied to 2 basic instructions of RISC-V are lui (load upper immediate) and addi (addition immediate) to load 32bits value.

A screenshot of a computer

Description automatically generated

After run the first step, the code runs the lui (load upper immediate) on the text segment window to load the 20bits value 0x20232 as most significant bits to s0 and it moves to next step to run addi (addition immediate) to load 12bits value 0x024 as least significant bits.

A screenshot of a computer

Description automatically generated

The results we get after run another step it’s the 0x20232024 on s0 in the value column of Registers window.

Similarly, we use li (load immediate) to load another value to s0 but the difference here that 0x20 it’s the 8bits value so it can run by using addi (addition immediate) so the complier use addi instead of using lui (load upper immediate) and addi like on loading 32bits value.

A screenshot of a computer

Description automatically generated

So the result we get on s0 in Registers window is 0x00000020.

**Assignment 4**

Source code:

A screenshot of a math equation

Description automatically generated

In this assignment, we use assembly to do the calculations. For example, we will calculate 2X + Y. We use addi (addition immediate) to load values to variables. In this example, we will store 5 to t1 and -1 to t2.

A screenshot of a computer

Description automatically generated

As we can see, after run 2 line addi (addition immediate), the value of t1 is 0x00000005 but the value of t2 is 0xffffffff. This because in a 32-bit system, the value 0xFFFFFFFF is the largest possible unsigned number. However, in signed systems, this same bit pattern represents -1 when using two's complement notation.

A screenshot of a computer

Description automatically generated

We do the calculation by using add (addition), it’s R-type instruction which is typically used for arithmetic and logic instruction. The result is simply just combining the value from X + X at line 3, we store at s0 and combine that with t2 is Y at line 4 and return it again on s0. The final result at s0 is 0x00000009.

**Assignment 5**

Source code:

A screenshot of a computer code

Description automatically generated

In this assignment, we use mul (multiple) to calculate Z = X \* Y. Different from other mathematical instruction, when multiplying two 32bits numbers, the result is a 64bits number. The RISC-V architecture provides some instructions to multiply that can write the result as 32bitsor 64bits. Those instructions are not included in RV32I architecture. It’s included in RV32M extension (RISC-V multiply/divided extension).

A screenshot of a computer

Description automatically generated

Assemble the code, we run 2 step by using addi (addition immediate) to load the value to t1 and t2. In this case, we assign t1 = 4 and t2 = 5 and the value return in t1 and t2 are 0x00000004 and 0x00000005.

A screenshot of a computer

Description automatically generated

After we run the next step, the program runs the mul (Multiply) instruction to multiply t1 and t2 and load the value to s1. From the image, we can see the value of s1 return is 0x00000014. However, we know that the value return to variable is hexadecimal. To check the result in the decimal we just need uncheck the Hexadecimal Values box.

A screenshot of a computer

Description automatically generated

As we can see here, the result of s1 is 20 in decimal so the calculation satisfies the problem we want to solve.

**Assignment 6**

Source code:

A screenshot of a computer program

Description automatically generated

In this assignment, we have introduced to declare variables. In this case, we declare variables X, Y, Z are the (4 byte) word type. Assign X = 5, Y = -1 and Z = 0. Declare variables by using compiler directives contain much information for compiling assembly code more accurately.

The directive .dataand .text works as bookmarks, locate the start address of a certain memory area in RAM, where the compiler will set the first variable or the first instruction. This starting point is purely a convention for controlling resources, so each CPU, each operating system, or compiler can set different starting points.

A screenshot of a computer

Description automatically generated

In the Labels window, the address of X is 0x10010000, Y is 0x10010004, Z is 0x10010008. So on the Data Segment window, on the first row, at the address 0x10010000, each column has 4 bytes apart. The Value +0 refers to the value of the X we have declared on .data is 5. Similarly with Y and Z, at Value +4 and Value +8, the value of 2 variables we declare at .data refers to the value of those 2 columns.

A screenshot of a computer

Description automatically generated

Instead of using addi (additional immediate), we use la (load address) to store the address of X and Y to t5 and t6. Same as the pointer in C language, we working with the variable through the address. The means of using this is that instead of manually calculating addresses or using multiple instructions to load a full address, we can just write la and let the assembler handle the details, the PC-relative addressing allows for writing code that can be moved around in memory without requiring changes to the addresses.

The la is the extended instruction so the assembler simulator needs to translate into two or more instructions depending on the target address size, at this case the address size of our variables are 32bits so it translates to 2 instruction auipc (Loads the upper 20 bits of the address relative to the program counter (PC)) and addi (Adds the lower 12 bits of the address to complete the full 32-bit address).

With the explanation, in the given image, we can see after run line 8 and line 9, the value of t5 and t6 are 0x10010000 and 0x10010004 respectively to the address of X and Y.

A screenshot of a computer

Description automatically generated

To load the word value from X and Y according to their address, we use lw (load word) to load the word value to t1 and t2.

The lw (Load Word) instruction is used to load a 32-bit word from memory into a register. This is a fundamental instruction for accessing data stored in memory. One thing we should notice is that the offset (Ex: 0(t5 or t6)). It’s a 12-bit signed immediate value that serves as the memory offset. The processor calculates the effective address by adding the immediate offset to the value in the base register. In this case, we refer the offset to the 0 to respect for the variable address.

Refer to Data Segment window, the value of X and Y are 0x00000005 (as 5) and 0xffffffff (as -1) have been loaded to t1 and t2 at the Registers window.

A screenshot of a computer

Description automatically generated

Similar with the assignment 4, we make the calculation Z = 2X + Y, the result is 9 and load the value of Z at s0 is 0x00000009.

A screenshot of a computer

Description automatically generated

Once again, we use la (load address) to load the address of Z to t4. We can see the t4 value is 0x10010008 refers to the address of Z.

A screenshot of a computer

Description automatically generated

To make sure that the value of Z is fixed, we use sw (store word) to store the word value to main memory.

The sw (store word) instruction is used to store a 32-bit word from a register into memory. This instruction moves data from the register file to a specified location in memory. Same as the lw (load word) we need to take an eye to the offset. In this case, we set the offset to 0(t4).

As the explanation, in the Data Segment window the value of Z in the Value +8 column refers to the address of Z (0x10010008), the value has been updated to 0x00000009.

For additional information, we have to make some explanations for lb (load byte) and sb (store byte).

1, The lb (load byte) instruction loads an 8-bit byte from memory into a 32-bit register, sign-extending the byte to fill the 32-bit register.

Syntax: lb **rd, offset(rs1)**

With:

* **rd**: The destination register where the loaded byte will be stored.
* **offset**: A 12-bit signed immediate value used as an offset.
* **rs1**: The base register that holds the starting memory address.

The processor calculates the **effective memory address** by adding the offset to the value in the base register rs1.

The byte at that memory location is loaded into the least significant byte of the destination register rd. The remaining bits of the register are filled by sign-extending the loaded byte

2, The sb (store byte) instruction stores the least significant byte (8 bits) from a 32-bit register into memory.

Syntax: sb **rs2, offset(rs1)**

With:

* **rs2**: The source register containing the value to be stored.
* **offset**: A 12-bit signed immediate value used as an offset.
* **rs1**: The base register that holds the starting memory address.

The effective memory address is calculated by adding the offset to the value in the base register rs1.

The least significant byte of the value in rs2 is stored into the memory location computed by the effective address.

So the difference between lb and sb is that:

* lb (load byte): Loads 8 bits from memory and sign-extends it to 32 bits.
* sb (store byte): Stores only the least significant 8 bits (1 byte) from a register into memory.

In Short:

* lb (Load Byte): Loads an 8-bit value from memory and sign-extends it into a 32-bit register.
* sb (Store Byte): Stores the least significant 8 bits of a 32-bit register into memory.

**Assignment 7**

Source code:

A screenshot of a computer

Description automatically generated

In this assignment, we declare the directives .data or .text with the address called literal address. The literal address is the start address of the memory area. The compiler will allocate variables or instructions from this address onwards.

The means of doing this is to avoid the changing of the address of the variable when we want to change the value of the variable. Same as variable, the address of the instruction is fixed.

Note:

* If you can find out the address of a variable of a target software, you can develop another software to illegally access that variable and change its value. That is hacking.
* If you can find the address of a instruction of a target software, you can develop another software to replace that instruction by a jump instruction to make CPU move to your codes. That is a computer virus.

A screenshot of a computer

Description automatically generated

After run the source code, we can see on the Labels window the address of x and y have been allocated and fixed. The address of X is 0x10011234 and Y is 0x10014320.

For the .text directive, we change the Combo Box under the Data Segment window to 0x00400000 (.text). We can see that at the row with the address 0x00408000, at the column Value +0, the value has been changed to 0x00200093, which refers to the code of the instruction “addi x1, zero, 2”. It’s shown that the address of the instruction has been fixed and the code of the instruction has been stored in the main memory.