LAB2 REPORT

Li Peihang

1 Checkoff 1

Trace through the execution of the code and compare the machine instructions and their corresponding source assembly instructions. Then answer the following:

(i) How many machine instructions have the `li` instruction been assembled into?

- If the constant immediate value is less or equal to 12 bits, the assembler translates the `li` instruction i it into 1 `addi` machine instruction:
  `li a1, 23 ->`
  `addi x11, x0,x23`

- If the constant immediate value is larger than 12 bits, the assembler translates the `li` instruction i it into 2 machine instruction:
  `li a2, 0x1234abcd ->`
  `lui x12, 0x0001234b`
  `addi x12, x12, 0xfffffbcd`

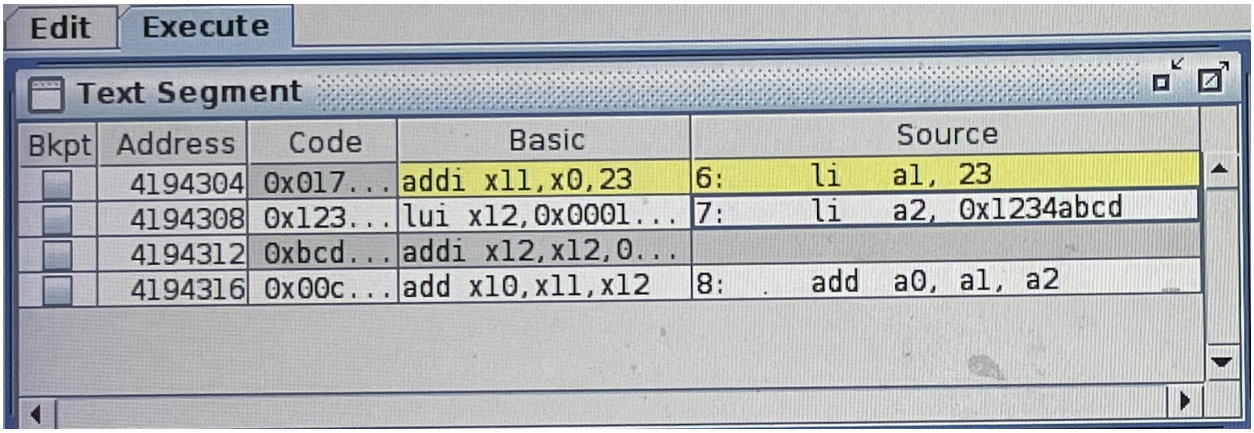

Figure 1: Assembly result by rars

(ii) Experiment with loading different constants. How are the li instructions being assembled differently with different constant values?

- if the number(immediate value) fits in `addi`'s immediate field ([-2048, 2047]), the assembler will translate the `li` pseudo instruction into just an `addi` instruction
  `li a0, immediate -> addi a0, x0, imme`

- otherwise, `li` will be translated into a `lui+addi` sequence

Let N be a 32-bit constant we want to load into a register: N = n31 ... n0. Then, we can split this constant into its upper 20 bits and lower 12 bits, NU and NL, respectively: NU = n31 ... n12 ; NL = n11 ... n0. In principle, we encode NU in the immediate in `lui` and NL in the immediate in `addi` [Notice: the `lui` is used to load unsigned immediate, the `addi` is used to load signed immediate].

- if the most significant bit of the 12-bit immediate in addi is 0

  ```
  li a0, immediate ->
  lui a0, (immediate >> 12)
  addi a0, a0, (immediate & 0xFFF)
  ```

- if the most significant bit of the 12-bit immediate in addi is 1

  ```
  li a0, immediate ->
  lui   a0,  ((immediate >> 12) + 1)
  addi  a0, a0, ((immediate & 0xFFF) - 2^12)
  ```

  If this is the case, the `addi` instruction adds to the destination register not NL, but NL - 4096 instead(-4096 is the resulting number when the upper 20 bits are 1s and the lower 12 bits are 0s).

  To compensate for the unwanted term -4096, we can add 1 to `lui` 's immediate – the LSB of the immediate in `lui` corresponds to bit 12 – so, adding 1 to this immediate results in adding 4096 to the destination register which cancels out the -4096 term.

- Specially, if the number larger than 12 bits and the lower 12 bits are 0, `li` will be translated into a `lui` instruction.

  ```
  li a0, immediate -> lui a0, (immediate >> 12)
  ```

lui rd, immediate — `x[rd] = sext(immediate[31:12] << 12)`

Load Upper Immediate. U-type, RV32I and RV64I.
Writes the sign-extended 20-bit *immediate*, left-shifted by 12 bits, to x[*rd*], zeroing the lower 12 bits.

Figure 2: lui instruction

addi rd, rs1, immediate — `x[rd] = x[rs1] + sext(immediate)`

Add Immediate. I-type, RV32I and RV64I.
Adds the sign-extended *immediate* to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

Figure 3: addi instruction

2 Checkoff 2

Trace through the execution of the file loadstore.s and answer the following questions:

(i) What are the values in the array arrayPrime after the code has completed?

11, 13, 24, 19

(ii) What is the base address of arrayPrime?

0x10010000

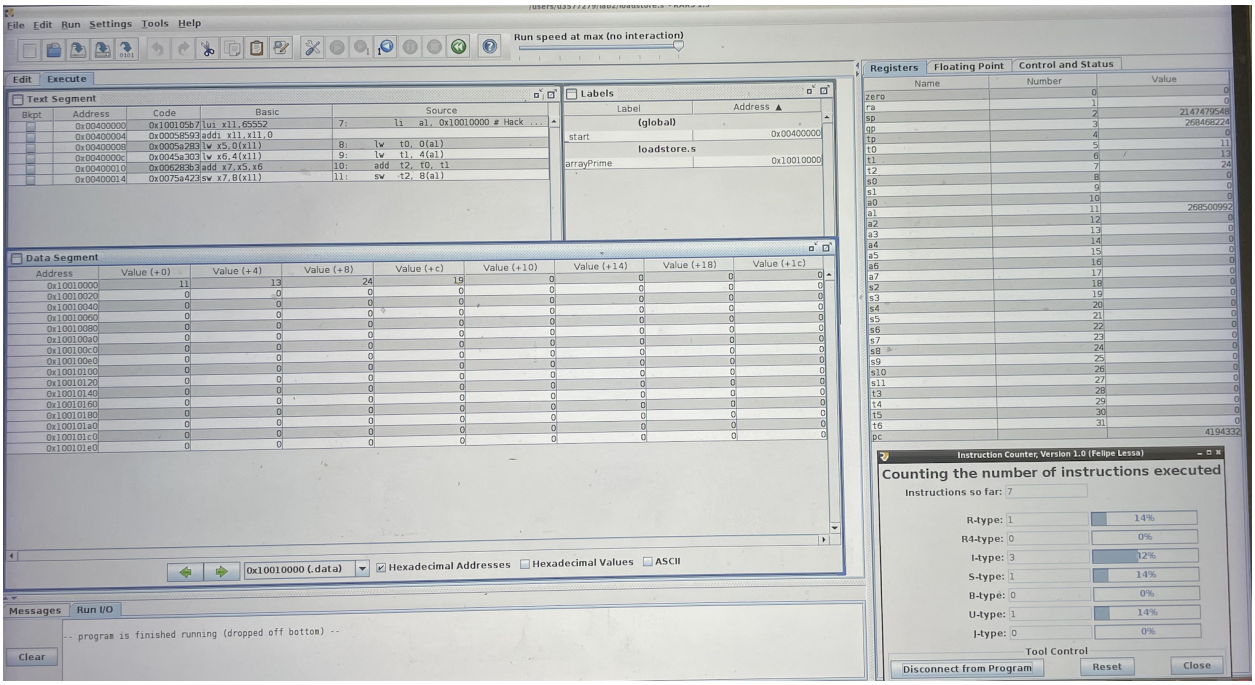

Figure 4: Results of loadstore.s using rars

(iii)If you insert another .word before the label arrayPrime and assemble the file again, where would the values of arrayPrime be moved to in memory?

I change the code to:

```
.data
.word  1000
arrayPrime:
.word  11
.word  13
.word  17
.word  19
```

The address of `arrayPrime` changes to `0x10010004` Thus, the values of arrayPrime are moved as follows in memory:

The first value (11) would be at `0x10010004`.

The second value (13) would be at `0x10010008`.

The third value (17) would be at `0x1001000C`.

The fourth value (19) would be at `0x10010010`.

Inserting a .word before an array or any label in assembly effectively shifts the starting address of that label and everything defined under it further in memory by the size of the inserted data (in this case, by 4 bytes for the .word insertion).

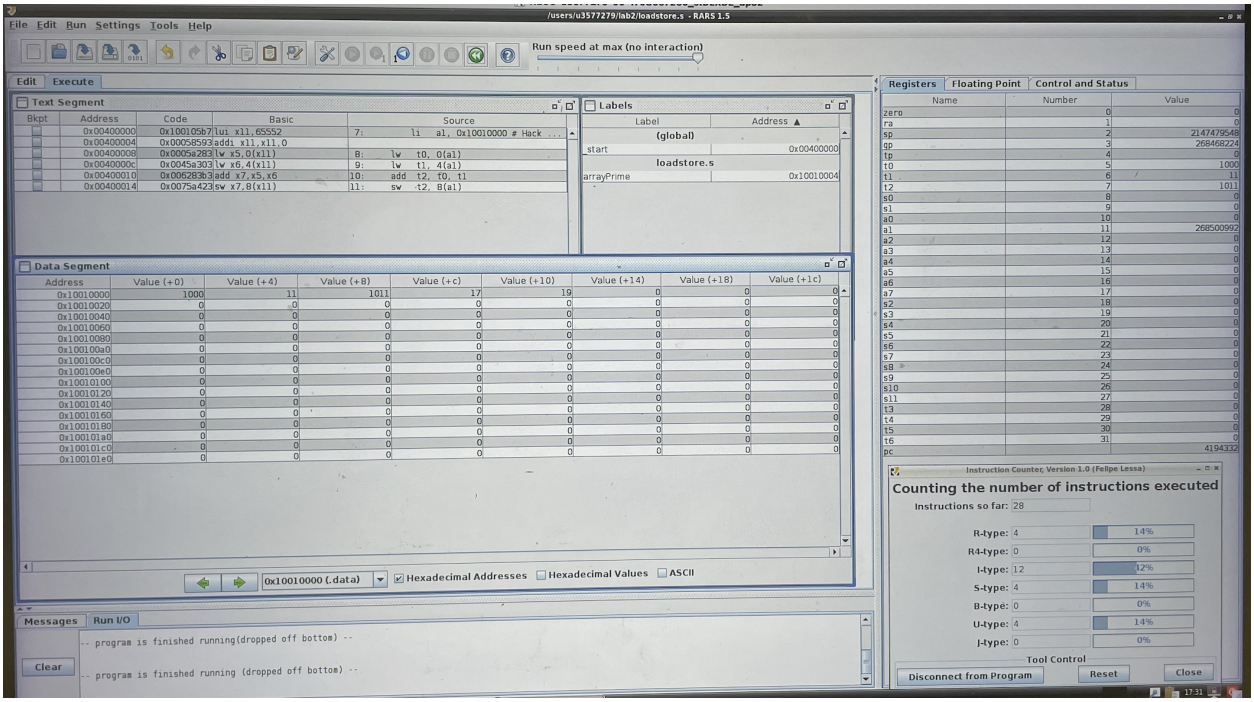

Figure 5: Result after insert another .word

3 Checkoff 3

(i) What is the value printed at the Run I/O pane after running the program?

3

(ii) In word, describe what is the function of the program mystery.s?

The program, therefore, serves to count the number of odd numbers in an array and display this count.

Verification From the values in the `anArray`, we have three odd numbers (29, 1337, and 0x0EEE3441) and three even numbers (0xABCD1234, 238, 1000). This aligns with the observed program output of 3, confirming the conclusion that the program counts the number of odd numbers in the array

Analysis

- Initialization: The program starts by loading addresses and preparing to iterate over an array. It calculates the end address of the array to know when to stop the loop.
- Loop and Summation Process:

  For each element in the array, it loads the element's value into `a0` and calls the `chk_number` function.

  The `chk_number` function performs an AND operation with `0x1` on the value in `a0`. This operation checks if the least significant bit of the number is set, which effectively determines if the number is odd (result is 1) or even (result is 0).

  It then returns immediately with this result in `a0`. The result from `chk_number` (which can be either 0 or 1) is added to `s0`, which accumulates the sum of these results. The loop continues until all elements have been processed.
- Output: After the loop, it prints a predefined string and the accumulated sum, which is the total number of odd numbers in the array (since each odd number would add 1 to the sum, and even numbers would add 0).