ALGORISK Documentation

Created by: Thibaud MARLIER
Creation Date: 09/02/2024
Last update: 20/02/2024

Copyright

Content

I. Introduction

Before You Read

Who should use this documentation?

The ALGORISK Assembly documentation is aimed at every user of our Assembler. It will help you throughout your journey of coding with our language and break down all the instructions for a deeper understanding of the ALGORISK language.

Chapter 1: Overview of the ALGORISK Assembler

Assembler Overview

Registers

The ALGORISK Assembly possesses multiple registers which can be grouped into 3 categories.

- Integer registers: from r1 to r16.
- Floating registers: from r17 to 32.
- Special registers:
    - r0: A constant register set to 0.
    - pc (program counter): Holds the address in memory of the next instruction to be fetched.
    - ir (instruction register): Holds the current instruction being executed.CSR (Control and status register): Holds the processor's configurations, it contains possible extensions of the instruction set, the information of the constructor, the architecture, and the implementation. They are hidden from the user.

Assembler Sections

Sections differentiate the variables set by the users from the actual code of the program.

ALGORISK possesses 2 sections:

Data Section

Data section: Contains the program's data, such as variables and constants.
The data section is determined by the .data directive and the declaration of a constant or a variable will be done like this:

```
.data
    myVariable: .type value
```

The different types of data will be:

- `.byte`: 8-bit signed integer
- `.half`: 16-bit signed integer
- `.word`: 32-bit signed integer
- `.float`: 32-bit floating-point number
- `.string`: null-terminated string
- `.alloc`: allocate the number of bytes specified by the following integer in memory.following integer in memory.

To declare arrays, you need to specify both the data type and the values for each element.

Examples:

```
.data
    myByte: .byte 255 // max value for a byte
    myHalf: .half 32767 // max value for a half
    myWord: .word 2147483647 // max value for a word
    myFloat: .float 3.402823466e+38 // max value for a float
    myString: .string "Hello, ALGORISK users!" // the max length depends on the
available space in memory
    myAlloc: .alloc // the max length depends on the maximum contiguous free space
in memory

    myArray: .word 1, 2, 3, 4, 5 // an array of 5 words
```

It's important to note that there are no explicitly unsigned types. The signedness of the data is often determined by the context in which it is used.
For instance, when loading or storing data using specific instructions, the signedness is inferred from the type of instruction being used (signed and unsigned instructions will be explained in the following part).
Always consider the context and the specific requirements of the instruction when working with different data types in ALGORISK assembly.

Code Section

Code section: Contains the program's instructions.
The code section is delimited by the .code directive. The declaration of a constant or a variable will be done like this:

where:

- rd means register destination, the register where the result will be stored.
- the number of parameters and their type may vary depending on the instruction.

```
.code
    [instruction] rd, parameter1, parameter2
```

Examples:

```
.code
    add r3, r1, r2
```

Chapter 2: Instruction Set

General Purpose Instructions

The general-purpose instructions perform basic data movement, memory addressing, arithmetic and logical operations, output, and string operations on integers and pointers.

In this documentation, you'll find registers which are defined as:

- rd = Destination Register
- r1 = Register number one
- r2 = Register number two

Data Transfer Instructions

ALGORISK instuction	Expanding	Description	Example
lb	Load Byte	Loads a signed byte from memory into a register, the address in memory must be specified as an operand	`lb rd, address`
lbu	Load Byte Unsigned	Loads an unsigned byte from memory into a register, the address in memory must be specified as an operand	`lbu rd, address`
lh	Load Halfword	Loads a signed halfword from memory into a register, the address in memory must be specified as an operand	`lh rd, address`
lhu	Load Halfword Unsigned	Loads an unsigned halfword from memory into a register, the address in memory must be specified as an operand	`lhu rd, address`
lw	Load Word	Loads a word from memory into a register, the address in memory must be specified as an operand	`lw rd, address`
lui	Load Upper Immediate	Loads an immediate value into the upper 20 bits of a register, the lower 12 bits are set to 0	`lui rd, immediate`
sb	Store Byte	Stores the lower 8 bits of a register into memory, the address in memory must be specified as an operand	`sb rd, address`
sh	Store Halfword	Stores the lower 16 bits of a register into memory, the address in memory must be specified as an operand	`sh rd, address`
sw	Store Word	Stores the lower 32 bits of a register into memory, the address in memory must be specified as an operand	`sw rd, address`

Binary Arithmetic Instructions

ALGORISK instuctions	Expanding	Description	Example
add	-	Adds the contents of two registers and stores the result in a register	`add rd, r1, r2`
addi	Add Immediate	Adds an immediate value to a register and stores the result in a register	`addi rd, r1, immediate`
sub	Subtract	Subtracts the contents of two registers and stores the result in a register	`sub rd, r1, r2`
mul	Multiply	Multiplies the contents of two registers and stores the result in a register	`mul rd, r1, r2`
mulh	Multiply High	Multiplies the contents of two registers and stores the upper 32 bits of the result in a register	`mulh rd, r1, r2`
mulhu	Multiply High Unsigned	Multiplies the unsigned value of two registers and stores the upper 32 bits of the result in a register	`mulhu rd, r1, r2`
mulhsu	Multiply High Signed Unsigned	Multiplies the signed value of a register with the unsigned value of another register and stores the upper 32 bits of the result in a register	`mulhsu rd, r1, r2`
div	Divide	Divides the contents of two registers and stores the result in a register (the destination register has to be from r16 to r31 to handle floats)	`div rd, r1, r2`
divu	Divide Unsigned	Divides the unsigned value of two registers and stores the result in a register (the destination register has to be from r16 to r31 to handle floats)	`divu rd, r1, r2`
rem	Remainder	Divides the contents of two registers and stores the remainder in a register (the destination register has to be from r16 to r31 to handle floats)	`rem rd, r1, r2`
remu	Remainder Unsigned	Divides the unsigned value of two registers and stores the remainder in a register (the destination register has to be from r16 to r31 to handle floats)	`remu rd, r1, r2`

Logical Instructions

The logical instructions perform basic logical operations on their operands.

ALGORISK instuctions	Expanding	Description	Example

ALGORISK instuctions	Expanding	Description	Example
and	-	Performs a bitwise AND operation on the values of two registers and stores the result in a register	`and rd, r1,r2`
andi	-	Performs a bitwise AND operation on the values of a register and an immediate and stores the result in a register	`andi rd, r1,immediate`
or	-	Performs a bitwise OR operation on the values of two registers and stores the result in a register	`or rd, r1, r2`
ori	Or immediate	Performs a bitwise OR operation on the values of a register and an immediate and stores the result in a register	`ori rd, r1, immediate`
xor	Exclusive or	Performs a bitwise XOR operation on the values of two registers and stores the result in a register	`xor rd, r1, r2`
xori	Exclusive or immediate	Performs a bitwise XOR operation on the values of a register and an immediate and stores the result in a register	`xori rd, r1, immediate`

Shift Instructions

ALGORISK instuctions	Expanding	Description	Example
sll	Shift left logical	Makes a logical shift of the bits of the first register to the left by the number of bits specified in the second register and stores the result in a register	`sll rd, r1, r2`
slli	Shift Left Logical Immediate	Makes a logical shift of the bits of the first register to the left by the number of bits specified in the second register and stores the result in a register	`slli rd, r1, immediate`
srl	Shift Right Logical	Makes a logical shift of the bits of the first register to the right by the number of bits specified in the second register and stores the result in a register	`srl rd, r1, r2`
srli	Shift Right Logical Immediate	Makes a logical shift of the bits of the first register to the right by the number of bits specified by the immediate and stores the result in a register	`srli rd, r1, immediates`
sra	Shift Right Arithmetic	Makes an arithmetic shift of the bits of the first register to the right by the number of bits specified in the second register and stores the result in a register	`sra rd, r1, r2`
srai	Shift Right Arithmetic Immediate	Makes an arithmetic shift of the bits of the first register to the right by the number of bits specified by the immediate and stores the result in a register	`srai rd, r1, immediate`

Conditional Instructions

ALGORISK instuctions	Expanding	Description	Example
ilt?	Is Less Than?	Compares the signed values of two registers, stores 1 if the first register is less than the second register, otherwise stores 0	`ilt? rd, r1, r2`
ilti?	Is Less Than Immediate?	Compares the signed value of a register with an immediate, stores 1 if the register is less than the immediate, otherwise stores 0	`ilti? rd, r1, immediate`
iltu?	Is Less Than Unsigned?	Compares the unsigned values of two registers, stores 1 if the first register is less than the second register, otherwise stores 0	`iltu? rd, r1, r2`
iltui?	Is Less Than Unsigned Immediate?	Compares the unsigned value of a register with an immediate, stores 1 if the register is less than the immediate, otherwise stores 0	`iltui? rd, r1, immediate`

Control Transfer Instructions

ALGORISK instuctions	Expanding	Description	Example
jie	Jump If Equal	Jumps to a label if two registers are equal	`jie r1, r2, label`
jine	Jump If Not Equal	Jumps to a label if two registers are not equal	`jine r1, r2, label`
jige	Jump If Greater or Equal	Jumps to a label if the signed value of the first register is greater than or equal to the signed value of the second register	`jige r1, r2, label`
jigeu	Jump If Greater or Equal Unsigned	Jumps to a label if the unsigned value of the first register is greater than or equal to the unsigned value of the second register	`jigeu r1, r2, label`
jile	Jump If Less or Equal	Jumps to a label if the signed value of the first register is less than or equal to the signed value of the second register	`jile r1, r2, label`
jileu	Jump If Less or Equal Unsigned	Jumps to a label if the unsigned value of the first register is less than or equal to the unsigned value of the second register	`jileu r1, r2, label`
jal	Jump And Link	Jumps to a label and stores the return address in a register	`jal rd, label`

ALGORISK instuctions	Expanding	Description	Example
jalr	Jump And Link Register	Adds an offset to a register and jumps to the address stored in the register, stores the return address in a register	`jalr rd, r1, offset`
auipc	Add Upper Immediate to PC	Adds an immediate value to the upper 20 bits of the program counter, the lower 12 bits are set to 0	`auipc rd, immediate`

Special Instructions

ALGORISK instuctions	Expanding	Description	Example
syscall	System Call	This transfers control to the operating system, and the system call handler performs the necessary actions (the syscall instruction does not take any operands)	`syscall`
break	-	Generates a breakpoint exception, which can be used for debugging	`break`

Index