RISC

Introduction

The final purpose of this document is not necessarily to be a technical documentation of the RISC built, but should also be instructional for other students interested in computer architecture. The goal is to design an efficient CPU with some intersting and modern features to learn their workings.

Instruction Set

Instructions

The instruction set consists of 32 different instructions. The first 5 bit of the operation code are interpreted as the instruction. The following instructions are implemented:

Data transfer	1	`LDW`	Load a value of a memory address into a register
	2	`LDB`	Load the LSB of a memory address into a register
	3	`MOV`	Move a value from one register to another
	4	`MOV`	Move an immediate value in a register
	5	`STW`	Store the value of a register in a memory adress
	6	`STB`	Store the LSB of a register value in a memory address
Stack	7	`PSH`	Push register values onto the stack
	8	`POP`	Pop values from the stack
ALU	9	`ADD`	Add the values of two registers
	10	`ADD`	Add an immediate value to a register
	11	`SUB`	Subtract the values of two registers
	12	`SUB`	Subtract an immediate value from a register
	13	`MUL`	Multiply the values of two registers
	14	`AND`	Bitwise AND of the values of two registers
	15	`ORR`	Bitwise OR of the values of two registers
	16	`XOR`	Bitwise XOR of the values of two registers
	17	`NOT`	Bitwise inversion of the value in one register
	18	`LLS`	Shift the value in a register left by a variable amount
	19	`ALS`	Shift the value in a register arithmetically left by a variable amount
	20	`RLS`	Rotate the value in a register left by a variable amount
	21	`LRS`	Shift the value in a register right by a variable amount
	22	`ARS`	Shift the value in a register arithmetically right by a variable amount
	23	`RRS`	Rotate the value in a register right by a variable amount
	24	`SXT`	Sign extension of the LSB
Branching	25	`BRX`	Branch to the address stored in a register
	26	`BIF`	Branch if a certain flag is set in the flag register

The operation codes have the following general layout:

Architecture

Operation Code	Mnemonic	Description
`0100 0001 0mmm 1nnn`	LDW	Load the word that is stored at the address held by the register `Rmmm` into the register Rnnn.
`0100 1001 0mmm 1nnn`	LDB	Load the least significant byte of the word at the address held by the register `Rmmm` into the register Rnnn.
`0101 0nnn bbbb bbbb`	MOVE	Move the value bbbb ' bbbb into the register Rnnn.
`0110 0001 0mmm 1nnn`	STRW	Store the value of the register `Rmmm` at the address held by the register Rnnn.
`0110 1001 0mmm 1nnn`	STRB	Store the least significant byte of the value in register `Rmmm` at the address held by the register Rnnn.
`0000 00ps xxxx xxxx`	PUSH	Push the declared registers x, the stack pointer s or the program counter p on the stack
`0000 10ps xxxx xxxx`	POP	Pop values from the stack in the declared registers x, the stack pointer s or the program counter p.
`1000 001m mmnn nddd`	ADD	Add the values of the registers `Rmmm` and Rnnn and store the result in the register Rddd.
`1000 101m mmnn nddd`	SUB	Subtract the value in the register Rnnn from the value in the register `Rmmm` and store the result in the register Rddd.
`1001 001m mmnn nddd`	AND	Calculate a bitwise AND of the values in the registers `Rmmm` and Rnnn. Store the result in the register Rddd.
`1001 101m mmnn nddd`	OR	Calculate a bitwise OR of the values in the registers `Rmmm` and Rnnn. Store the result in the register Rddd.
`1010 0001 0mmm 1ddd`	NOT	Bitwise inversion of the value in the register `Rmmm`. Store the result in the register Rddd.
`1010 1001 0mmm bbbb`	LSHIFT	Shift the value in the register `Rmmm` by bbbb to the left.
`1011 0001 0mmm bbbb`	RSHIFT	Shift the value in the register `Rmmm` by bbbb to the right.
`1011 1000 0000 1nnn`	SIXT	Signextend the least significant byte of the value that is stored in the register Rnnn.

ALU

The ALU consists of the following components:

- Brent-Kung Adder (BKA)
- Barrel Shifter
- Dadda Tree Multiplier (DTM)
- Logic Unit (LU)

Brent-Kung Adder

A Brent-Kung adder is a parallel prefix adder. It has a layer depth of $\mathcal{O}(\log_2(n))$ for an input size of n.

Barrel Shifter

The barrel shifter is able to (arithmetically) shift and rotate all the bits to the left or to the right. The design was taken from Pillmeier et al. [1]. They found, that the delay of the shifter increases with $\mathcal{O}(\log(n))$ and the area with $\mathcal{O}(n \log(n))$ as the operand's size n grows.

Dadda Tree Mulitplier

A Dadda Tree Multiplier uses a Dadda Tree to multiply two numbers.

References

[1] Matthew R. Pillmeier, Unisys Corporation, Michael J. Schulte, E George, and Walters Iii. Design alternatives for barrel shifters. *Proceedings of SPIE - The International Society for Optical Engineering*, 4791, 10 2002.