



Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors:

Alessandro Loschi, Andrea Mongardi

October 29, 2018

Contents

1	DLX Behaviour	1
1.1	Instructions	1
1.2	Pipeline	2
1.3	Instruction Set	2
1.4	Datapath	2
1.4.1	Pipeline implementation	2
1.5	Control Unit	2
1.6	2
2	More in details	4
2.1	ALU	4
2.1.1	Adder	4
2.1.2	Multiplier	4
2.1.3	Divider?	4
2.1.4	Logic	4
2.1.5	Comparator	4
2.2	Simulations	4
2.3	Synthesis	4
2.4	Layout	4
A	Adder behavioural VHDL	5

CHAPTER 1

DLX Behaviour

The DLX is a RISC microprocessor able to do basic operations of this category. The purpose of this project is to implement a DLX-like processor, with some additional characteristics. We start giving a general description of this device and how it works. Then, we will go deep in our project.

1.1 Instructions

Instruction format is on 32 bit and we have a different 6-bit opcode for each one. Depending on this code, we can have 3 different types of instructions:

R-Type: For this kind of instruction, the datapath is configured using op-code and func, to make alu register to register operations.

This type of instructions are characterized by the format:

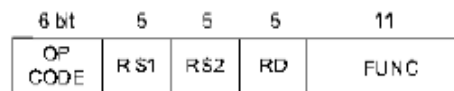


Figure 1.1: R-Type format

I-Type: they are load and store instructions, operations with immediates or conditional branches. The format here is:

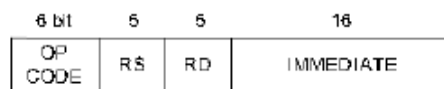


Figure 1.2: I-Type format

This operations involve immediates or conditional branches, plus conditional load and store.

J-Type: They are jump instructions and have a format:



Figure 1.3: J-Type format

1.2 Pipeline

The pipeline is composed of 5 different stages(Clock Cycles):

Instruction Fetch: During this stage the Program Counter is updated, and the corresponding instruction is loaded from the instruction memory into the instruction register.

Instruction Decode/Register Fetch: The instruction is decoded and registers A,B and IMM are fed by the register file.

Execution: The values stored in the registers from the previous stage are processed by the alu. The result is stored into ALUOut register.

Memory Access/Branch Completion: Load/Store data from/into the data memory into LMD or coming from ALU. In branches, the PC is replaced with the destination address in the ALUOut register.

Write-Back: Write results into the register file.

1.3 Instruction Set

We implement all the basic DLX instructions, and we add a set of other instructions in order to move our project to pro. The table 1.1 shows the complete list.

1.4 Datapath

Our datapath is divided into 5 different units, each one implementing a pipeline stage.

1.4.1 Pipeline implementation

The general architecture is:

1.5 Control Unit

Is the component in advance of send/receive signals from/to datapath in order to manage the instruction flow in the correct way. We use to choose this an hardwired CU, rather than others, because...

1.6

Mnemonic	Coding	Mnemonic	Coding	Mnemonic	Coding	Mnemonic	Coding
J	J,0x02	SRAI	I,0x17	SLEUI	I,0x3C	SGE	R,0x2D
JAL	J,0x03	SEQI	I,0x18	SGEUI	I,0x3D	SLTU	R,0x3A
JR	J,0x04	SNEI	I,0x19	SLL	R,0x04	SGTU	R,0x3B
JALR	J,0x05	SLTI	I,0x1A	SRL	R,0x06	SLEU	R,0x3C
BEQZ	B,0x06	SGTI	I,0x1B	SRA	R,0x07	SGEU	R,0x3D
BNEZ	B,0x07	SLEI	I,0x1C	ADD	R,0x20	MULT	F,0x0E
ADDI	I,0x08	SGEI	I,0x1D	ADDU	R,0x21		
ADDUI	I,0x09	LB	L,0x20	SUB	R,0x23		
SUBI	I,0x0A	LH	L,0x21	SUBU	R,0x24		
SUBUI	I,0x0B	LW	L,0x23	AND	R,0x25		
ANDI	I,0x0C	LBU	L,0x24	OR	R,0x26		
ORI	I,0x0D	LHU	L,0x25	XOR	R,0x27		
LHI	I,0x0F	SB	S,0x28	SEQ	R,0x28		
XORI	I,0x0E	SH	S,0x29	SNE	R,0x29		
SLLI	I,0x14	SW	S,0x2B	SLT	R,0x2A		
NOP	N,0x15	SLTUI	I,0x3A	SGT	R,0x2B		
SRLI	I,0x16	SGTUI	I,0x3B	SLE	R,0x2C		

Table 1.1: Instruction Set

CHAPTER 2

More in details

2.1 ALU

2.1.1 Adder

2.1.2 Multiplier

2.1.3 Divider?

2.1.4 Logic

2.1.5 Comparator

2.2 Simulations


2.3 Synthesis

2.4 Layout

APPENDIX A

Adder behavioural VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ADDER32 is
    generic (n: NATURAL:= 33); 
    port (INA_ADD: in signed (n-1 downto 0);
          INB_ADD: in signed (n-1 downto 0);
          OUT_ADD: out signed (n-1 downto 0);
          CTRL_ADD1, CTRL_ADD2: in STD_LOGIC);
end entity ADDER32;

architecture Behaviour if ADDER32 is
    begin
        discrimination: process (CTRL_ADD1, CTRL_ADD2) is
            begin
                if CTRL_ADD1 = '1' and CTRL_ADD2 = '0' then
                    OUT_ADD <= INA_ADD - INB_ADD;
                elsif CTRL_ADD1 = '0' and CTRL_ADD2 = '1' then
                    OUT_ADD <= INB_ADD - INA_ADD;
                else
                    OUT_ADD <= INA_ADD + INB_ADD;
                end if;
            end process;
        end process;
    end architecture Behaviour;
```