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

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:

6 bit	5	5	5	11
OP CODE	RS1	RS2	RD	FUNC

Figure 1.1: R-Type format

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

6 bit	5	5	16
OP CODE	RS	RD	IMMEDIATE

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.

Instuction 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 Completition: 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 Instruction Set.

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

1.4 Datapath

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

- Fetch Unit;
- Decode Unit;
- Execution Unit;
- Memory Unit;
- Write-back Unit;

1.4.1 Pipeline implementation

The general architecture, including all units, is:

Every unit/stage is separated by dashed lines. Stages are connected in cascade, the order is like in the list above.

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 choose to use an hardwired CU, rather than others, because...

1.6 Memories

We use two RAM as Instruction and Data memories. The IRAM is able to acquire instructions from a compiled .asm file, with the correct coding (see appendix A for the VHDL code).

CHAPTER 2

More in details

2.1 ALU

The core of all operations is the ALU, collocated in the execution unit. It is the component in charge of doing logical and arithmetical operations. The ALU is configured externally by the C.U., selecting which is the function.
It is composed of :

- Adder;
- Multiplier;
- Logic;
- Comparator;

2.1.1 Adder

The architecture of our adder is like the P4 one implemented during laboratories. We choose this configuration to avoid high carry delays and to make the sum faster. The general architecture is:

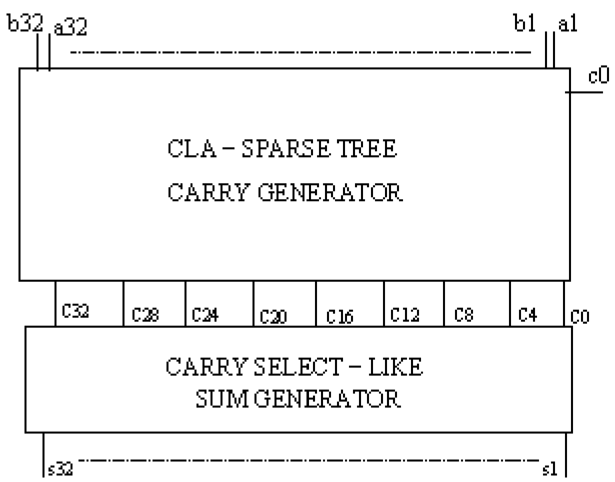

Figure 2.1: P4 Adder schematic.

The two blocks are in charge of doing a sum or a subtraction. The idea is to compute partial carries and propagate them into the sum generator, reducing the computational time w.r.t. the traditional Ripple Carry Adder. Obviously, to obtain the configuration for the subtraction, the second input B is xored with Cin.

Carry generator The sparse tree carry generator architecture is shown below:

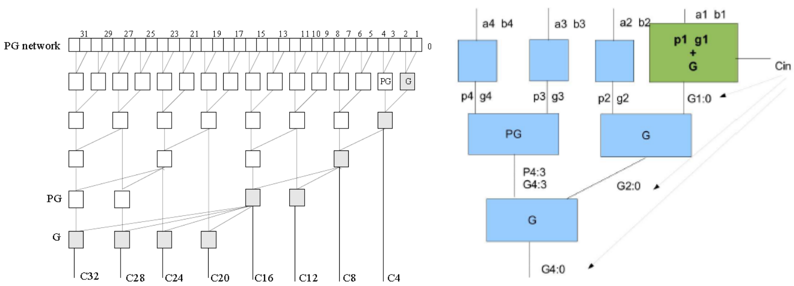

Figure 2.2: P4 Adder Carry generator and details.

G and PG blocks implement general Generate and Propagate blocks, defined as:

$$G_{i:j} = G_{i:k} + P_{i:k} * G_{k-1:j}; \quad (2.1)$$
$$P_{i:j} = P_{i:k} * P_{k-1:j}; \quad (2.2)$$

where

- i $\geq$ k $>$ j;
- $G_{x:x} = g_x$ that is the generate term and $P_{x:x} = p_x$ that is the propagate term;
- g_0 = Cin and $p_0 = 0$;
- $g_i = a_i * b_i$;
- $p_i = a_i + b_i$;

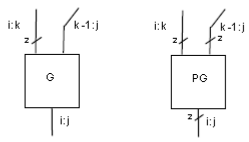

Figure 2.3: Block G and PG.

The first G−block generate only $G_{i:j}$ and the other PG−block generate both $G_{i:j}$ and $P_{i:j}$.

Sum Generator This block is a Carry−Select Adder, each subblock use a Ripple Carry Adder for partial sums;

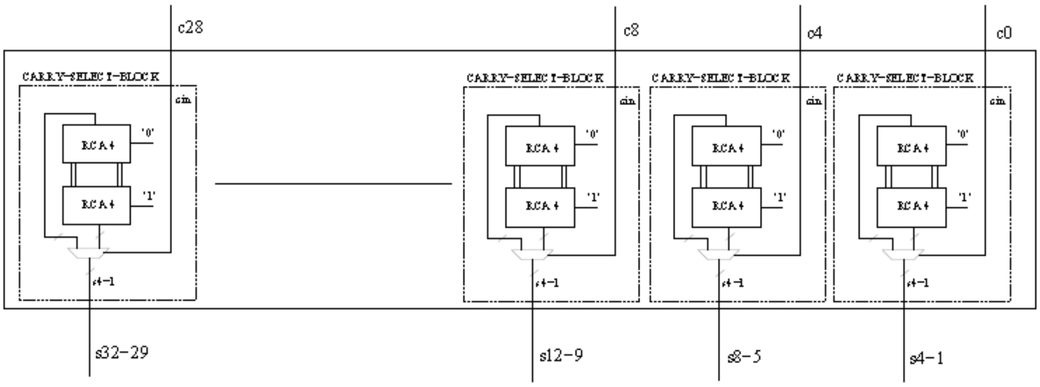

Figure 2.4: Carry Select Adder with Carries coming from sparse tree.

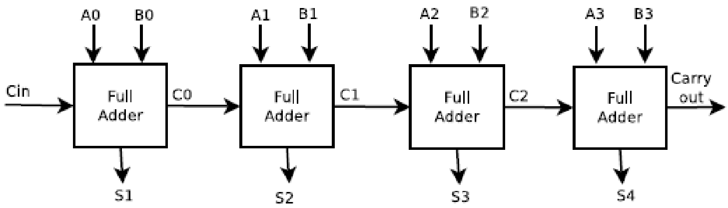

Figure 2.5: 4-bit RCA inside p4 adder.

2.1.2 Multiplier

2.1.3 Logic

We implement a simple way to do logic operations. The operands pass through 32 parallel gates bit by bit, implementing the requested operation. We choose this configuration in order to have the same delay for all bits of operands, even if it results in a large area.
Examples:

- $\text{ALUOut}_i <= \text{A}_i$ and B_i;
- $\text{ALUOut}_i <= \text{A}_i$ or B_i;

2.1.4 Comparator

We use the comparator for conditional instructions. We implement a classic architecture, using an adder in subtraction configuration and gates. The architecture is the following:

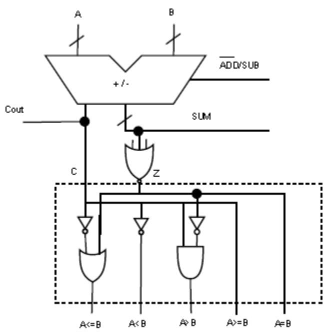

Figure 2.6: Comparator.

2.2 Simulations

2.3 Synthesis

2.4 Layout

APPENDIX A

IRAM VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.logarithm.all;


-- Instruction memory for DLX
-- Memory filled by a process which reads from a file
-- file name is "test.asm.mem"
entity IRAM is
	generic (	RAM_DEPTH	: integer := 48;
			I_SIZE		: integer := 32);
	port (	Rst  : in  std_logic;
		Addr : in  std_logic_vector(RAM_DEPTH - 1 downto 0);
		Dout : out std_logic_vector(I_SIZE - 1 downto 0));
end IRAM;

architecture IRam_Bhe of IRAM is

	type RAMtype is array (0 to 2**RAM_DEPTH - 1) of integer;

	signal IRAM_mem : RAMtype;

begin  -- IRam_Bhe

	Dout <= conv_std_logic_vector(IRAM_mem(conv_integer(unsigned(Addr))),I_SIZE);

	-- purpose: This process is in charge of filling the Instruction RAM with the
	   firmware
	-- type    : combinational
	-- inputs  : Rst, Addr
	-- outputs: IRAM_mem
	FILL_MEM_P: process (Rst)
		file mem_fp: text;
		variable file_line : line;
		variable index : integer := 0;
		variable tmp_data_u : std_logic_vector(I_SIZE-1 downto 0);
	begin  -- process FILL_MEM_P
		if (Rst = '0') then
			file_open(mem_fp,"test.asm.mem",READ_MODE);
			while (not endfile(mem_fp)) loop
				readline(mem_fp,file_line);
				hread(file_line,tmp_data_u);
				IRAM_mem(index) <= conv_integer(unsigned(tmp_data_u));
				index := index + 1;
			end loop;
		end if;
	end process FILL_MEM_P;
end IRam_Bhe;
```