Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: Group 15

Alberto Anselmo, Giulio Roggero

Wednesday 31st October, 2018

Contents

CHAPTER 1

Adders

1.1 Full Adder

Adders are one of the most used digital components in computer processors. In general an adder is a digital circuit that implements the sum of two numbers expressed on N bits.
In particular, a full adder (FA) is characterized by three inputs and two outputs. If we consider a one-bit full adder, the three inputs are the two numbers to sum and the input carry. The outputs are the sum and the output carry. A schematic diagram of a one-bit full adder is shown in figure

The truth table of the one-bit full adder is shown in table 1.1.

A B Cin	S	Cout
0 0 0	0	0
1 0 0	1	0
0 1 0	1	0
1 1 0	0	1
0 0 1	1	0
1 0 1	0	1
0 1 1	0	1
1 1 1	1	1

Table 1.1: Truth table of a 1-bit full adder.

From the truth table we can easily derive the logic equations that describe the full adder:

$$S = A \oplus B \oplus C_{in} \tag{1.1}$$

$$C_{out} = AB + C_{in}(A \oplus B) \tag{1.2}$$

1.1.1 Area and power estimation

A simple Bash script has been written to automatise the calculus of area and power starting from the layout of the circuit. The script takes in input the SVG (which is the primary Inkscape's format) description of the layout and it extracts all the necessary information. The output of the script is a .txt file containing different data.
The output text file of the full adder is reported below.

```
Reference layout = planar_full_adder.svg
Number of layers = 1
Total number of magnets = 150
Clock zones = 5
Width [nm] = 2080
Number of vertical magnets =       11
Height [nm] = 1300
Area [nm2] = 2704000
Area [um2] = 2.704000
########### Cu wire case ############
Wire 1 resistance [Ohm] = .26010000000000000000
Wire 2 resistance [Ohm] = .32500000000000000000
Power consumption for one layer [W] = .00000526590000000000
Total power consumption [W] = .00001053180000000000
########### Ta wire case ############
Wire 1 resistance [Ohm] = 81.12642857142857142857
Wire 2 resistance [Ohm] = 101.36904761904761904761
Power consumption for one layer [W] = .00002919927619047619
Total power consumption [W] = .00005839855238095238
```

To summarize, the planar full adder is characterized by:

- Delay = 3 clock cycles;
- Area = 2.7 μm^2;
- Power = 10.53 μW.

1.2 Ripple Carry Adder

A Ripple-carry adder (RCA) is a more complex adder composed by a cascade of more full adders. It is used to add N-bit numbers and its name derives from the fact that the carry propagates from a full adder to the next one.

. . . .

. . . . . .

. . .

CHAPTER 2

Fetch Stage

2.1 Introduction

The fetch stage is important to provide to our processor a new instruction at every clock cycle, if needed. However, in this part of the pipeline, there are different components.

CHAPTER 3

Execution Stage

3.1 Overview

We have designed an execution unit that is capable of dealing with almost all the instructions present in the instruction set that has been given. Exceptions are the floating point operations, which requires speicific hardware that we decided not to include. It is worth mentioning that normally the *MUL* operations are executed by dividing them in subsequently pipelined stages. Here, differently, in order to keep the 5 stages in the pipeline and not to alter the normal execution flow, this doe not take place. As a consequence, the maximum operating frequency will be heavily affected by the propagation time needed by the multiplier to finish its job.

In the *Execution Unit* are therefore present the following components:

- **shifter**, used in order to shift and rotate the value contained in an input register
- **adder**, necessary obviously to perform addition and subtraction between values, but also used in the logic comparison operations
- **logic**, unit to perform logic bitwise operations, such as *OR, AND, XOR* and so on
- **comparator**, which purpose is to determine whether an input is greater, smaller or equal than another one
- **multiplier**, whose goal is to perform multiplications between inputs
- PC adder, a dedicated adder to increase by the value of the PC, useful in case of a *JMP* operation
- **registers**, to update the inputs and store the results
- **muxes**, to select the results coming from the required unit and correctly update the outputs of the stage

- **inverter**, needed to perform the *2'S complement* and perform the subtraction

For this stage, a bunch of bits of the CW are used. Here are reported:

- *ENEX*, a general enable for the components and especially for the registers
- *MUX1_SEL*, a selection signal for the mux that is able to select an immediate value or a register an input that as the first term for the other units
- *MUX2_SEL*, with a goal similar as the one above, but for the second input
- *UN_SEL*, 3 bits signal that selects the correct input, which is then sent to a register
- *OP_SEL*, 4 bits selection signal, used to indicate to each unit, with different encoding, the operation to be performed
- *PC_SEL*, selection signal to update the PC counter correctly, following a branch or jump.

3.2 Adder

For the adder architecture, we decided not to start from scratches. As long as our P4 adder, designed during the laboratories, has a general description, we resized it to our architecture on 32 bits. This means that we have included the components capable of producing the PG propagate and generate signals and the G ones used to produce only the generate and so the final carry. We consider that therefore the result, in terms of performance can satisfy us. At the input of the block, we have also a *C_IN* signal, which is used to create, in cooperation with the inverter, the 2'S complement of the input used (either immediate or coming from a register), in order to produce as a final result the signed sum. There are few control signals used for this unit. In figure 3.1 is presented the simple schematic of the adder block.

3.3 Logic

In order to perform some logic bitwise operations, we have reproduced the microarchitecture that has been used for the T2. Here, in figure 3.2 is briefly reported the gates needed, that combined between each other, are capable of giving us the desired operations, as reported in the table 3.1.

The control signal S is required to be on 4 bits: for this reason, we have implemented in our control word 4 bits for this purpose. In particular, the signal OP_SEL plays this role. We could have used 3 bits on the *CW* and design an encoder, but the savings in term of connection would have been ruined by the need of an additional decoder. Moreover, these bits are not switched most of the time, thus providing a not so relevant switching power dissipation.

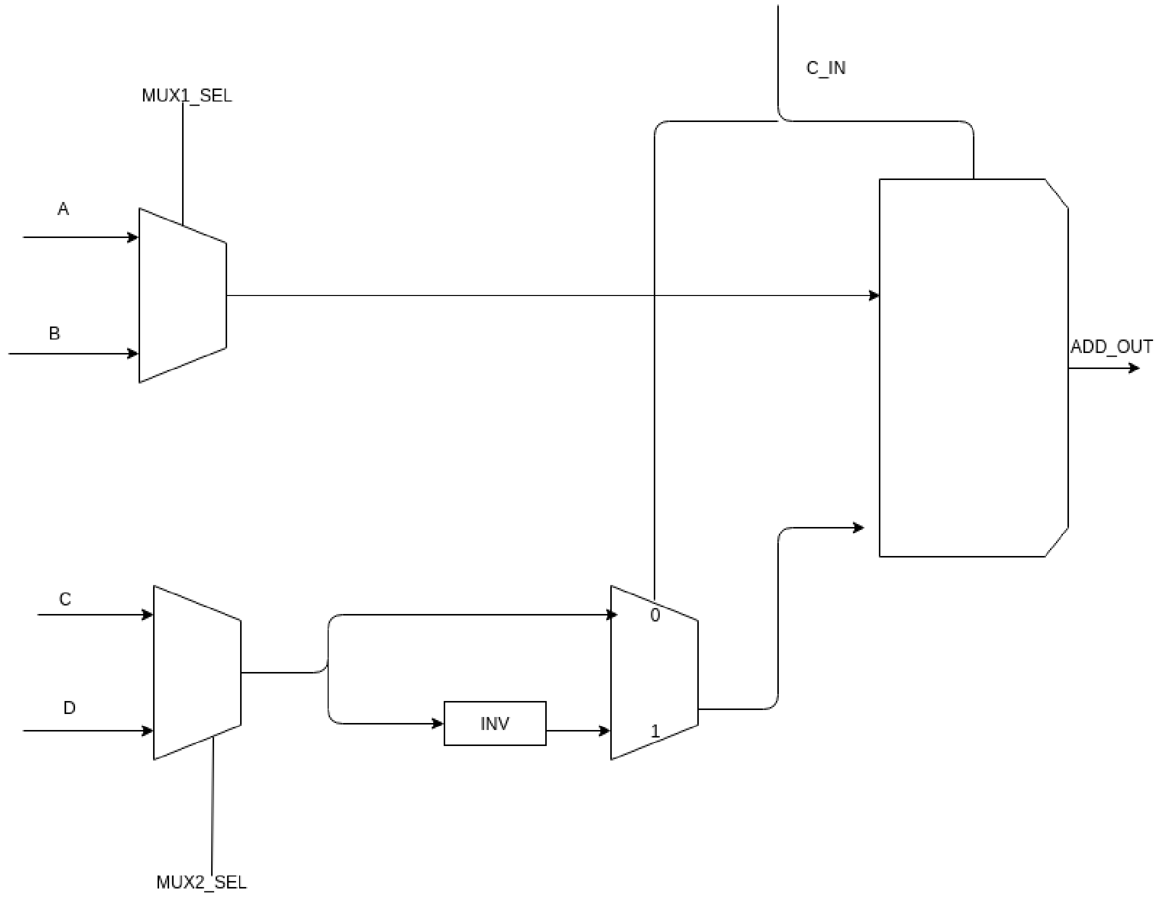

Figure 3.1: Adder with inverter and related selection signal

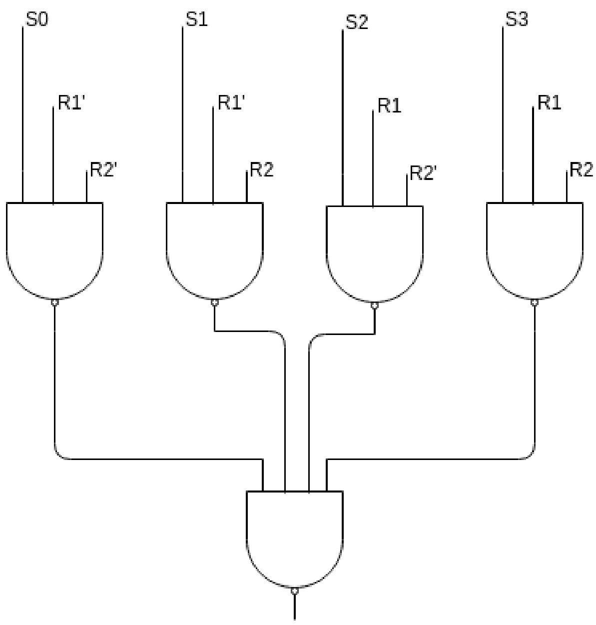

Figure 3.2: T2 logic for a single bit

	S0	S1	S2	S3
AND	0	0	0	1
NAND	1	1	1	0
OR	0	1	1	0
NOR	1	0	0	0
XOR	0	1	1	0
XNOR	1	0	0	1

Table 3.1: Combination of control signal for logic operations

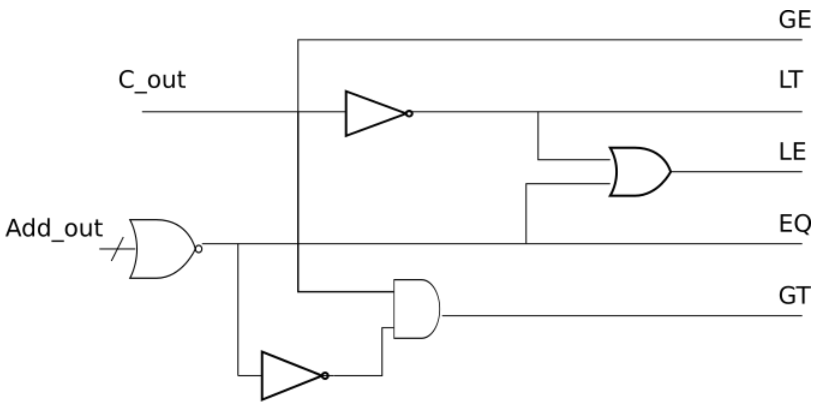

Figure 3.3: Circuit for comparison - Unsigned

3.4 Shifter

To define the lateral shifts, left or right, and the rotation, we have simply provided a behavioral description of the circuit, as reported in appendix A. It generates a barrel shifter, thus providing the needed operation between the ones possible, and moving the bits of the first input by the quantity that is introduced by the second input.

3.5 Logic comparison

The logic comparison requires the output of the subtraction between the two values to be compared. So, *ADD_OUT*, output of the adder, is here used as an input. Then, all the bits of the result are *NOR_reduced*. This allows us to see if all the bits are equal to 0, i.e. the two operands are equal. In this case, an additional signal *US* is used, to differentiate between an unsigned or a signed comparison. While in the first case the schematic is the one presented in figure 3.3, for the signed case we have to differentiate. As a matter of fact, if the first digit is equal, the two numbers have the same sign, so we can use the same HW presented before. Differently, if the two bits differ, we use what is reported in 3.4. The equivalence of the two MSBs is performed through a XNOR gate.

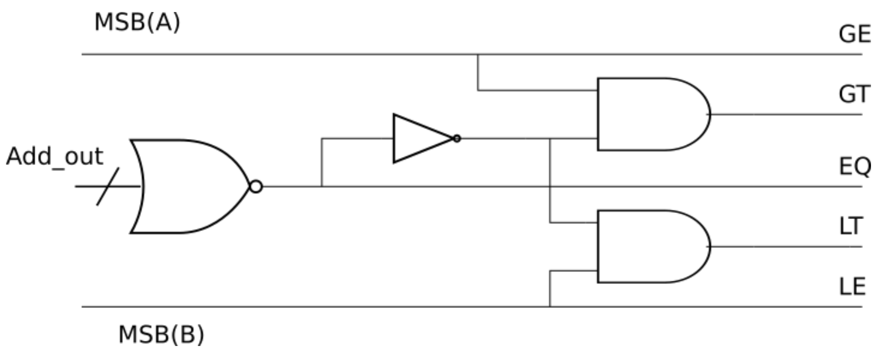

Figure 3.4: Comparison for signed case - different MSBs

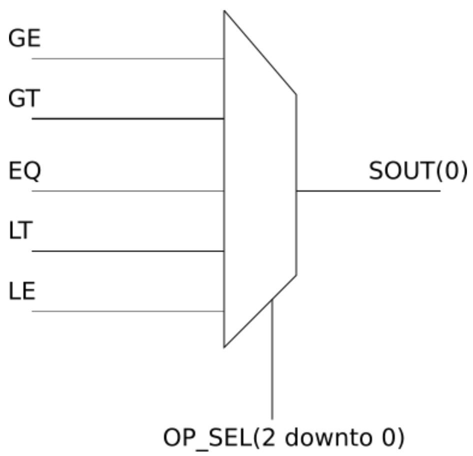

Figure 3.5: Mux to select the required condition

Followingly, the value corresponding to the logic comparison needed is chosen by a mux, whose selection signal is represented by *OP_SEL*, and assigned to the lowest bit of the output signal *SOUT*, as shown in figure 3.5. All the other bits of this signal are independetly set to 0.

3.6 Multiplier

For our multiplier we again reused the *Booth* architecture that has been designed for the laboratories. As said, the multiplication is usually supposed to be divided in more than one stage, but here it works on a single one, thus reducing the operating frequency. However, we decided to include this component in order to include the MUL in our instruction set.

3.7 PC adder

In order to perform correctly some jump instruction, the temporary PC needs a +4, which, due to the design of our datapath, can not be performed by the adder. An additional adder, reported in figure 3.6 is in charge of this specific operation.

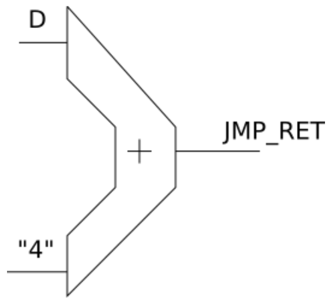

Figure 3.6: PC adder

CHAPTER 4

Memory Stage

This stage can be divided in two major parts: the registers that contains the addresses and the data to be written in the memory or read from the memory and the RAM. The former is actually embedded in the processor, while the latter one is actually external, but provides an interface and some signals to communicate with the processor itself.

4.1 Embedded part

The part included in the datapath is mainly made up of three different registers, which are used to store different values coming from different units:

- *dest_reg*, which is used to save the address of the commit register, to be passed to the following write back stage
- *mem_reg*, where the data coming from the external memory presented in 4.2 are stored
- *exec_reg*, which is used to save the address of the commit register, to be passed to the following write back stage

All these registers share the same reset, clock and enable signals. They differ for the input/output ones, which are summarized in table 4.1.

Reg Name	# bits	Input	Output
exec_reg	32	FROM_ALU	ALU_OUT
mem_reg	32	FROM_MEM	MEM_OUT
dest_reg	5	DEST_IN	DEST_OUT

Table 4.1: Table with registers of Memory Stage

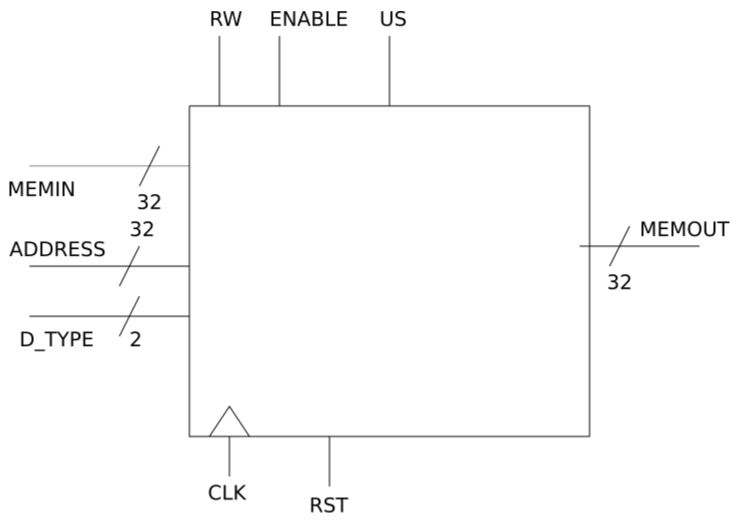

Figure 4.1: Interface for external RAM

4.2 External RAM

The external RAM receives signals both from the datapath and from the CU. An overview can be seen in figure 4.1. This part is external since it will not be synthesized. In particular, here are presented the most relevant ones:

- *ENABLE, RW* are signals used to control the writing on the memory. They should be both high to complete a write
- *D_TYPE* specifies the kind of data we are reading or writing in/from memory, i.e byte, half-word, word
- *US* defines a signed/unsigned representation for the data; it is relevant to correctly extend the data on 32 bits
- *ADDRESS* obviously used to correctly point to a precise memory location
- *MEMIN* is the signal containing values to be eventually stored in memory
- *MEMOUT* is the output signal which cotains the data read from the memory

While the writing is performed synchronously only if the *RW* signal is high, there is always a value in correspondence of the output signal. This means that a read always happens, and a value is written every time on the *MEMOUT* signal. This value will be eventually discarded in the following stages.

CHAPTER 5

Write Back

The *Write Back* stage is needed to write in the registers the final result obtained after the execution and the access to memory. With this goal, an destination address is needed, as well as some muxes to select the right signal to be sent back to the RF. The overall structure is presented in figure 5.1. Few signals are needed in the entire unit:

- **MEM_ALU_SEL**, selection signal coming from the control word, controlling the output mux
- **DEST_IN**, destination address from the memory stage
- **FROM_ALU**, coming directly from the execution stage
- **FROM_MEM**, that comes from the memory, after a read operation has been performed
- **DATA_OUT**, which contains the data to be written on the Rf
- **DEST_OUT**, destination address to the RF

To simply explain the behavior, one can say that the destination address is contained in the 5 bits of *DEST_IN*, which are passed to *DEST_OUT*. This output signal is then connected to the RF, where the *WR* signal enables the writing. The actual value that is written on memory is chosen between the outputs from memory and ALU, and ultimately written to close the pipeline and complete the instruction, which is finally committed.

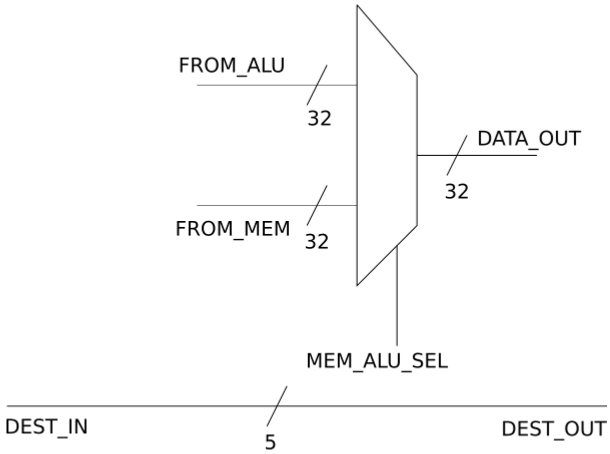

Figure 5.1: Mux to choose the required output

CHAPTER 6

Additional features

6.1 Data forwarding

6.2 Dynamic branch prediction

In order not to waste too much clock cycles waiting for the correct value of the PC register, we have designed a unit that provides a dynamic prediction regarding a branch instruction. In particular, it is possible, with a defined value of the PC corresponding to a branch, to get whether is a better speculation to take or not to take the branch, while waiting for the correct PC, which comes after the execution stage. There are

APPENDIX A

Shifter behavioural VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;


entity SHIFTER is
  generic (NB: integer := 32;
                    LS: integer:= 5
                    );
  port    (     FUNC:                    IN std_logic_vector(1
     downto 0);
                    US:                          IN std_logic;
                DATA1:                   IN std_logic_vector(NB-1
                    downto 0);
                DATA2:                   IN std_logic_vector(LS-1
                    downto 0);
                OUTSHFT:                 OUT std_logic_vector(NB-1
                    downto 0)
          );
end SHIFTER;

architecture BEHAVIOR of SHIFTER is


begin
```

```
P_ALU: process (FUNC, US, DATA1, DATA2)
  begin
        if(US='1') then
            case FUNC is
                -- SLL
                    when "00"      =>
                                   OUTSHFT <=
                                       std_logic_vector
                                       (shift_left(
                                       unsigned(DATA1),
                                        to_integer(
                                       unsigned(DATA2))
                                       ));
                    -- SRL
                    when "01"      =>
                                   OUTSHFT <=
                                       std_logic_vector
                                       (shift_right(
                                       unsigned(DATA1),
                                        to_integer(
                                       unsigned(DATA2))
                                       ));
                    -- ROL
                    when "10"      =>
                                   OUTSHFT <=
                                       std_logic_vector
                                       (rotate_left(
                                       unsigned(DATA1),
                                        to_integer(
                                       unsigned(DATA2))
                                       ));
                    -- ROR
                    when "11"      =>
                                   OUTSHFT <=
                                       std_logic_vector
                                       (rotate_right(
                                       unsigned(DATA1),
                                        to_integer(
                                       unsigned(DATA2))
                                       ));
```

```
                    when others =>
                        OUTSHFT <= (others =>'0');
            end case;
        else
                case FUNC is
                    -- SLL
                    when "00"         =>
                                      OUTSHFT <=
                                          std_logic_vector
                                          (shift_left(
                                          signed(DATA1),
                                          to_integer(
                                          unsigned(DATA2))
                                          ));
                    -- SRL
                    when "01"         =>
                                      OUTSHFT <=
                                          std_logic_vector
                                          (shift_right(
                                          signed(DATA1),
                                          to_integer(
                                          unsigned(DATA2))
                                          ));
                    -- ROL
                    when "10"         =>
                                      OUTSHFT <=
                                          std_logic_vector
                                          (rotate_left(
                                          signed(DATA1),
                                          to_integer(
                                          unsigned(DATA2))
                                          ));
                    -- ROR
                    when "11"         =>
                                      OUTSHFT <=
                                          std_logic_vector
                                          (rotate_right(
                                          signed(DATA1),
                                          to_integer(
                                          unsigned(DATA2))
```

```
                                                    ));
                                    when others =>
                                        OUTSHFT <= (others =>'0');
                            end case;
                    end if;
  end process P_ALU;

end BEHAVIOR;
```