Politecnico di Torino

Microelectronics Systems

DLX microprocessor: Design & Development

Final project report

Group 24

Francesco Ricci
Master Degree in Electronics Engineering
Master in science in Embedded Systems

Lorenzo Zaia
Master degree in computer science
Master in science in Embedded Systems

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Contents

1 Introduction **3**

2 Instruction Set **4**

3 Datapath **6**
- 3.1 Fetch stage . . . 6
  - 3.1.1 Dependency manager . . . 6
  - 3.1.2 Synchronization . . . 7
  - 3.1.3 Compiler . . . 7
- 3.2 Decode stage . . . 8
  - 3.2.1 Jump-Logic . . . 8
  - 3.2.2 Jump address computation . . . 8
  - 3.2.3 Decision . . . 8
  - 3.2.4 Flushing . . . 8
- 3.3 Execute stage . . . 8
  - 3.3.1 P4 adder . . . 9
  - 3.3.2 T2 logic unit . . . 9
  - 3.3.3 Shifter . . . 9
  - 3.3.4 Comparator . . . 10
  - 3.3.5 Booth's multiplier . . . 10
- 3.4 Memory access stage . . . 10
- 3.5 Write back stage . . . 10

4 Control Unit **12**
- 4.1 ALU opcode . . . 12
- 4.2 Registers enables . . . 12

5 Synthesis **14**
- 5.1 Preliminary steps . . . 14
- 5.2 Synthesize a clocked block . . . 14
- 5.3 Timing report . . . 14

6 Layout **15**
6.1 Vdd and Gnd routing 15
6.2 DLX gates placement 15
6.3 Pin positioning 15
6.4 Routing 15
6.5 Optimizations 15
6.6 Analysis 16

Chapter 1

Introduction

The goal of this project is the design of a "DLX" microprocessor. The design can be divide into four main steps:

- **Simulation**: a VHDL description of the pipeline is simulated several times, and adjusted until correct behaviour is achieved
- **Synthesis**: boolean laws (like DeMorgan) and complex algorithms are exploited to simplify/optimize the electronics, producing a generic gate-level architecture
- **Mapping**: Generic gates function are searched for in the specific target technology library. Optimized gates are found and constraints are taken into account thanks to particular algorithms
- **Layout**: Decides position of physical interconnects between inputs and cells, among cells and finally between cells and outputs. Decides metal layer for each connection. Particular care for Vdd, Gnd, Clock routing

The DLX microprocessor is based on the RISC architecture. The main principles are low number of instructions, fixed length instructions and simple addressing modes (immediate and displacement). The outcome is a much simpler and efficient hardware, and slightly bigger code.

Our DLX processor is capable of executing four classes of instructions: R-TYPE, I-TYPE, J-TYPE and L/S-TYPE.

Chapter 2

Instruction Set

The following instructions have been successfully implemented:

- R-TYPE:
  - Arithmetical
    - ADD
    - SUB
    - MULT
  - Logical
    - AND
    - OR
    - XOR
  - Comparison-based
    - SGE
    - SLE
    - SNE
  - Shift-based
    - SLL
    - SRL
    - SRA
- I-TYPE:
  - Arithmetical
    - ADDI
    - SUBI
    - MULTI
  - Logical
    - ANDI
    - ORI
    - XORI
  - Comparison-based
    - SGEI
    - SLEI
    - SNEI
  - Shift-based
    - SLLI
    - SRLI
    - SRAI
- J-TYPE
  - BEQZ
  - BNEZ
  - J
  - JAL
- S/L-TYPE
  - LW
  - SW

Our DLX is capable of executing also those extra instructions:

- NOR
- XNOR
- NAND
- NORI

- `XNORI`
- `NANDI`
- `SEE`
- `SEEI`
- `RL`
- `RR`
- `RLI`
- `RRI`

However, because the provided compiler does not included those instructions, we tested them by using a free OPCODE and writing a code using the corresponding instruction, even if in reality it implements a totally different function.

Chapter 3

Datapath

The datapath is pipeline-based to ensure a optimal trade off between power and timing(every clock cycle an instruction is completed). The pipeline used in our DLX is composed of five stages:

- `Fetch`
- `Decode`
- `Execute`
- `Memory Access`
- `Write-Back`

3.1 Fetch stage

The fetch stage is conceptually "external" from the datapath: has the job of loading the next instruction from the instruction memory (IRAM) into the instruction register (IR) by looking at the current value of program counter register (PC). The IR content is sent to the Control Unit (CU) and to the fetch stage. The CU is in charge of computing and delivering a Control Word. A control word is a collection fo control signals that adjust the datapath behaviour, in order to perform the required operation.

- `Program Counter register (PC)`: pointer to the current instruction being executed.
- `IRAM`: RISC architecture has 2 different memories: the instruction memory contains the code that should be executed. When reset is on, the code is loaded into the IRAM.
- `Adder`: a very simple/non optimized adder perform NPC = PC+4.
- `Dependency Manager`: finds dependencies between the instructions inside the pipeline and the current instruction that is about to be executed. **If dependency is present, one or multiple stalls are introduced into the pipeline.**
- `Connection with CU`: the IR is propagated to the control unit.
- `Boundary registers`: the fetch stage receive the value of NPC, SOURCE1, SOURCE2, DESTINATION and IMMEDIATE(26 bits).

3.1.1 Dependency manager

Let's take a deep look into how the dependency manager works. The working principle:

- by looking at the current instruction opcode, understand if the instruction has 2 source registers (RTYPE) or 1 source register (ITYPE, LW, BRANCHES) or no source registers at all (JUMPS)
- look at the destination registers of instructions already inside the pipeline
- compare those destinations with the source(s) of your current instruction.
- If at least one of the sources in equal to any destination, it means a RAW dependency is present,

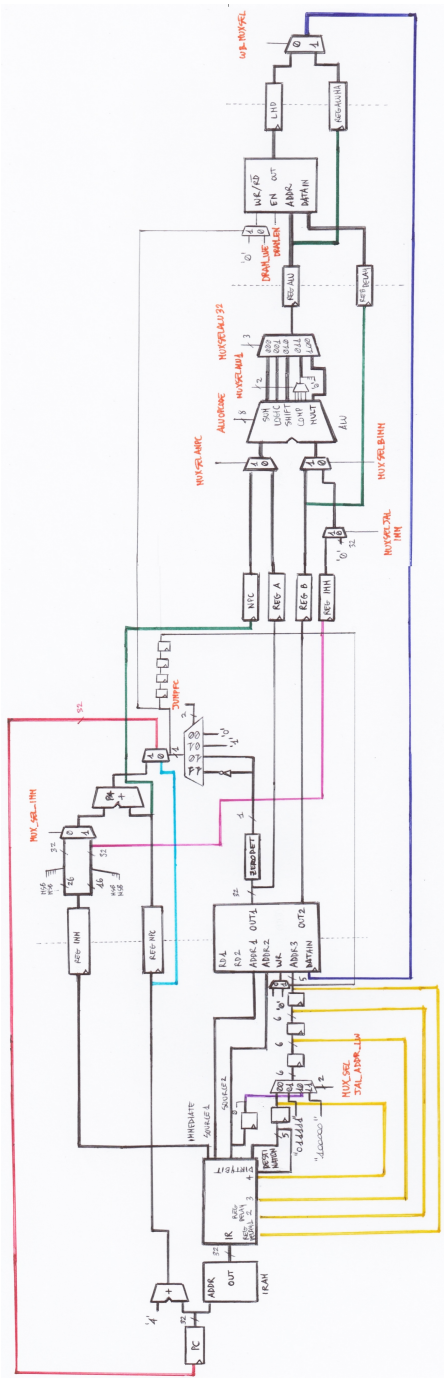

Figure 3.1: Datapath schematic

and stalls should be issues into the pipeline. Also stop PC from increasing.

In practice, the output of each register inside the register chain (see write back stage for details) is compared to source(s) of the current instruction. If a comparison is positive, a single stall (NOP operation) is inserted inside the pipeline, and an enable signal is switched off. This signal is connected to the PC register and the NPC, IMMEDIATE registers between fetch and decode stage. After a clock cycle, the register chain will be composed by a NOP and other 3 destinations. The comparison is executed again: if the destination causing the dependency is still inside the register chain, another stall is issued. And so on until, in the worst case 4 NOPS are issued.
This implementation works fine until you consider register R0. In fact, R0 and NOP are the same (00000). This maybe cause a deadlock! However, by introducing a dirty bit, the problem is solved.
The register chain is composed by 6bit registers (dirty bit + 5bits for the address). If a NOP is issued, the register chain will receive 100000 (dirty bit = 1), otherwise rity bit remains = 0. The register file considers only the LSB 5 bits (and discards the dirty bit).

3.1.2 Synchronization

In order to update the program counter (PC) each clock cycle (if no dependencies are present), a combinational path connecting the output of the PC register to the input of the same register was essential. This path is represented in the figure 3.1 (light blue connection). When a jump is taken, the instruction after the jump should be flushed (see jump logic for more details).

3.1.3 Compiler

A compiler (coded in perl) was provided. By using this compiler we were able to convert assembly instructions into 32 bits sequence of zeros and ones.
By restarting the simulation, the new instructions were loaded into the IRAM.

3.2 Decode stage

The IR contains the addresses of the registers that will be used as operands (sources or destination). Their current value is read from the register file, and transmitted to the next stages.
In our DLX the jump decision was moved from execute stage into the decode stage:

- ADV: 1 clock cycle saving when flushing the pipeline (jump/branch taken)
- DIS: extra adder is required. Increasing area and power consumption.

Inside this stage:

- `Register-File`: characterized by 2 read ports and 1 write port.
- `Jump-logic`: multiplexers, a zero detector and an adder. Jump and branch instructions are executed correctly.
- `Boundary Registers` : the execute stage receive the NPC (the same received by the decode), A, B, IMMEDIATE (32 bits).

3.2.1 Jump-Logic

Let's take a deep look into how the jump logic is organized. Three main sections can be recognized:

- `Jump address computation`
- `Decision`
- `Flushing`

3.2.2 Jump address computation

A jump or a branch require an immediate value (26 bits for jumps, 16 bits for branches) to be added to the NPC, to compute the new address. For this reason, the last 26 bits from the IR are extended (arithmetical,to allow also backward jumps) into 32 bits. The same is done considering the last 16 bits of IR. The control unit chooses among which extended immediate should be added to the NPC, by controlling a multiplexer. An addition between an immediate and NPC in done, and the result goes into another multiplexer that choose between the new address and the NPC (actually NPC coming directly from fetch). This means that every clock cycle PC is updated and when a jump is taken, the previous instruction in the pipeline should be flushed.

3.2.3 Decision

Our DLX provide two conditional branches: BEQZ, BNEZ. For this reason, the out1 of the RF is connected also to a zero detector. The result goes into a 4to1 multiplexer.
The inputs of this mux are 0 (not taken), 1 (taken = jump), eq (BEQZ), not(eq) (BNEZ).
This multiplexer is controlled by the CU, so the right signal is choosen each time. The output is connected as selection signal of the last multiplexer described in the previous section.

3.2.4 Flushing

In order to flush the instruction after the jump (if the jump/branch is taken), our DLX will execute the instruction, but will not store the result in DRAM or RF.
To achieve this goal, the jump-sel signal is propagated using 2 flip flops to the memory access stage. It controls a multiplexer, that chooses between the CU value for WR/RD in of DRAM and zero (no WR). The same is done for RF, but to ensure synchronization 4 flip flops are used.

3.3 Execute stage

The operands are combined and the result is computed. The stage is composed by the ALU and some multiplexers, that select the correct inputs, and the requested output. The ALU is composed by the following sub-blocks:

- `P4 adder/subtractor`: compute 32-bits addition and subtractions.
- `T2 logic unit`: compute AND, NAND, OR, NOR; XOR, XNOR.

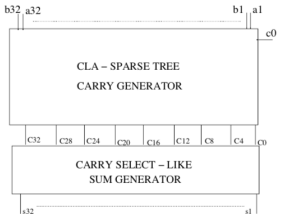

Figure 3.2: Adder two main components

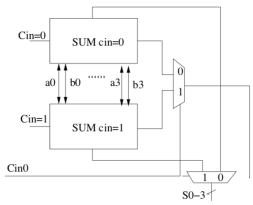

Figure 3.3: carry select adder

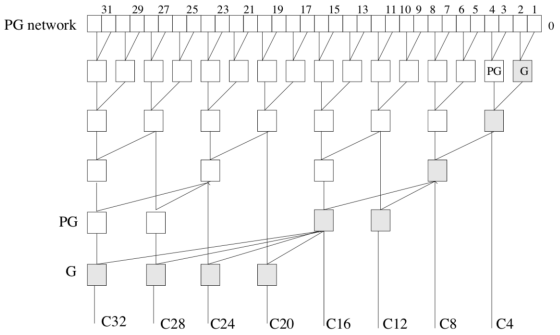

Figure 3.4: Sparse tree

- `Shifter`: compute shifts and rotations.
- `Comparator`: compute $=, !=, >=, >, =<, <$. It use adder output.
- `Booth's Multiplier`: performs 16 bits*16bits multiplications.

`Boundary Registers`: the memory access stage receive the ALU's result and the B operand (just delayed by one clock cycle, to maintain synchronization).

3.3.1 P4 adder

The P4 adder is composed by two main components (figure 3.2):

- sum generator: carry select adder (CSA)
- carry generator: sparse tree

Sum generator

The principle of the P4 adder is to avoid the long propagation delay of the carry in a RCA (ripple carry adder).
In our DLX implementation, each input value (32 bits) is divided into 8 pieces, each one of 4 bits. For each 4 bit couple (A and B) the sum is performed using two different RCA, one with carry in set to 0, the other with carry in = 1. This pre-computation speeds up the process, because the right sum is selected by looking at the carry generator output. A detailed view is reported in figure 3.3.

Carry generator

The sparse tree exploit the definition of generate (a and b) and propagate (a xor b). This structural computes in a combinational way the carry every 4 bits. A detailed view is reported in figure 3.4.

3.3.2 T2 logic unit

The T2 logic unit is defined in figure 3.5. It is made only by NAND gates (each one is implemented as a tree on 2-inputs NANDs). There are 4 control signals, and the right combinations of them selects the logic function. The advantages of this approach is a really small area, and a balanced path length, no matter which logic function is implemented.

3.3.3 Shifter

The shifter is the one provided inside the project materials. It is characterized by left/right, arithmetical/logical shifts. It is also capable of doing rotations.

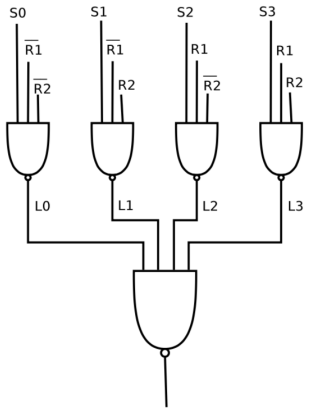

Figure 3.5: T2 logic unit

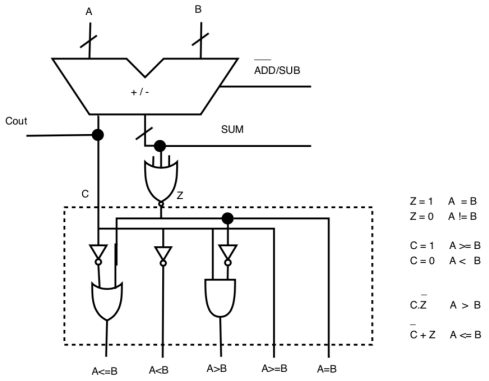

Figure 3.6: Comparator

3.3.4 Comparator

The comparator implemented is our DLX exploit the subtractor present inside the ALU.
The working principle is the following: subtract the two inputs and look at result and at the carry out.
A detailed view is reported in figure 3.6.

3.3.5 Booth's multiplier

Taking advantages from Booth's Algorithm implementation, this multiplier is faster (it has a reduced number of partial sums) and much more area efficient (compensation algorithm) with respect to the classical array multiplier.

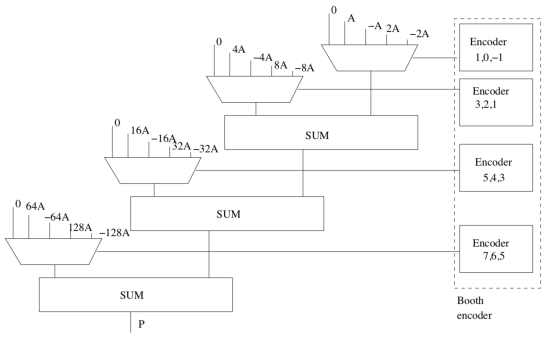

Figure 3.7: Booth's multiplier

Since, in our DLX, the multiplier output is on 32 bits, to avoid overflow, the maximum length of the operands must be 16 bits.
A detailed view is reported in figure 3.7.
In our DLX, there are 8 multiplexers, 7 adders (with different parallelism) and 1 encoder.

3.4 Memory access stage

The result can be stored directly inside the data memory (DRAM) if necessary. Composed by:

- `DRAM`: has register B has data-in, and the ALU result as address. It can perform load and store operations.
- `Boundary Registers`: the write back stage receive the ALU result (just delayed by one clock cycle, to maintain synchronization) and the DRAM output (if a load has been performed).

3.5 Write back stage

The result can be stored into the register file. It is the simplest stage, because it is composed by:

- `Multiplexer`: chooses among ALU result and DRAM output
- `Register chain`: in order to perform the write back in the correct location of the register file,

the write address of RF has been delayed by 4 clock cycles using a chain of registers.

Chapter 4

Control Unit

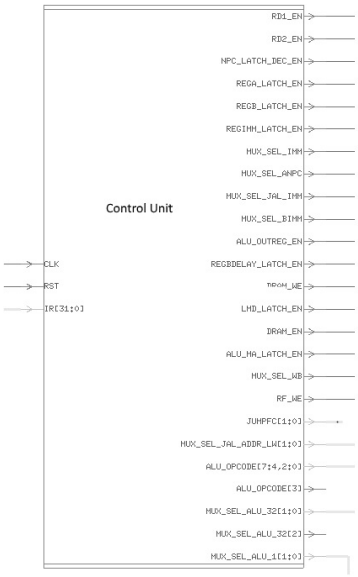

Figure 4.1: CU top view

A control unit can be implemented as hardwired, micro-programmed or by using a FSM. In our project, the CU is hardwired. The goal of a control unit is compute and deliver the correct portion of control word to each stage. For this reason the CW is divided in four sub-control words, each one delayed by one clock cycle with respect to the previous one. This delay insertion is necessary to respect the intrinsic flow of the pipeline architecture.

A B C D	function
0 0 0 1	and
1 1 1 0	nand
0 1 1 1	or
1 0 0 0	nor
0 1 1 0	xor
1 0 0 1	xnor

Table 4.1: Sequence for each logic function

4.1 ALU opcode

The ALU inside our DLX is capable of executing additions, subtractions, shifts, rotations, logic functions and multiplication. The OPCODE is divided as follow:
3bits shift, 1 bit adder, 4 bits logic functions

Shifter:
first bit: logical (1) or arithmetical (0);
second bit: left (1) or right (0);
third bit: shift (1) or rotation (0).
Adder: set to 0 for addition, 1 for subtraction.
Logic functions: see table 4.1 for details.

4.2 Registers enables

The DLX is capable of executing different types of instructions. We introduced in our CU basically enable signals for each register inside the datapath. By keep-

ing a register disabled when is not used, the power consumption is reduced.

Chapter 5

Synthesis

5.1 Preliminary steps

Before starting the synthesis phase, the IRAM has been removed from the DLX. For this reason the top_entity has now two additional ports: PC and IR. This choice was made to reduce the synthesis overhead.

5.2 Synthesize a clocked block

During the synthesis phase, the clock must be taken in account. For this reason at the beginning, a clock period should be selected. To add the clock to the synthesis process, the following command should be included into the script:

create_clock -name CLK -period X CLK

where X is the selected period.
Consequentially, inside the synthesis script, a constraint about the maximum delay between two memory elements should be enforced. The command to include into the script is:

set_max_delay X -from [all_inputs] -to [all_outputs]

where X is the selected period.

5.3 Timing report

The best clock cycle period can be selected by looking at the timing report.
On the first try, we selected a clock period of 2 ns. After the synthesis, the timing report contained a negative slack (-1.56 ns). This means that at least one path had a longer delay with respect to the clock period.
To solve this issue, a lazier clock should be used. The second try was 3.6 ns (2ns + 1.56 ns). The timing report showed a positive slack this time (1.03 ns). This indicated that inside the range between 2 and 3.6 ns the best possible clock cycle can be found.
After several additional tries we choosed the the clock period of 2.2 ns. This choice is still characterized by a positive slack, but much reduced (0.27 ns). Is important to maintain a little bit of margin, because during the layout phase, some delays can possibly be increased.

Chapter 6

Layout

The layout phase is divided into 6 steps:

- Vdd and Gnd routing
- DLX gates placement
- Pin positioning
- Routing
- Optimizations
- Analysis for time and integrity

6.1 Vdd and Gnd routing

Vdd and Gnd are critical signals: they should be delivered across the entire circuit in the most reliable way possible. For this reason dedicated rings are designed, and the highest two metal layers are mostly reserved for those signals.
In our DLX, two power rings are inserted around the die area, using metal layers 9 and 10.
Several stripes are routed vertically, each one separated from the other by 20 micrometers. In our DLX 11 stripes were placed.
Finally,starting from the stripes, a dense net is created, in order to reach every possible spot on the chip. This step is called "cell power routing".

6.2 DLX gates placement

The cadence tool uses the verilog file containing the complete synthesized design, to extract and place every gate inside the available space.

6.3 Pin positioning

Our DLX has 4 inputs:

- CLK and RST are just 1 bit each,
- PC, IR are 32 bits vectors.

The IRAM has not been synthesized, so PC is the address used to drive the IRAM and IR is its output (the current instruction that should be executed).
Top and bottom of the chip were reserved to clock and reset, right side present the the PC pins, left side presents the IR pins.

6.4 Routing

This step is crucial: all connections from cell to cell, and from cells up to Vdd, Gnd, inputs pins are routed. The Innovus tool performs several steps in order to optimize the routing. It provides also some interesting data like: the number of wires used for each layer, the total length routed in each layer, the number of via used.

6.5 Optimizations

Two main optimizations has been performed: after placement and after routing. They both involves the setup time and hold time optimization. Other steps

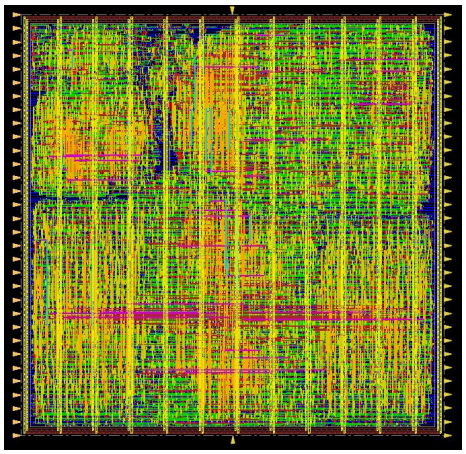

Figure 6.1: DLX after layout phase

like placing filler gates (gates always off, that will provide mechanical strength to the circuit). At this point the layoutis complete, as shown in figure 1.

6.6 Analysis

Three main analysis are performed:

- Parasitics: for each metal wire, Innovus calculate the capacitance and the resistance and uses them to perform a timing analysis.
- Report Timing provides reports containing detailed information on the timing paths and violations. The slack is a good indicator to understand if there is a violation.
- Verification: to evaluate integrity, Innovus provides tools for checking the geometry and connectivity.