Politecnico di Torino

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

Microelectronic systems

DLX Project

Renzi Alessandro 19778

July 21, 2013

Contents

1 Introduction

The goal of this project was to design a pipelined DLX microprocessor at RTL level, synthesize it, and do the physical design.
Moreover optimize it and do a post physical performance analysis, I imposed to myself two constraints for the design: a clock frequency of 100 MHz and a power consumption of 2 mW.

2 Architecture

The architecture of this microprocessor is the classical DLX with five stage pipeline plus some variations.
The stages are:

- IF: Instruction fetch
- ID: instruction decode
- EX: Execution
- MEM: Memory
- WB: Write-Back

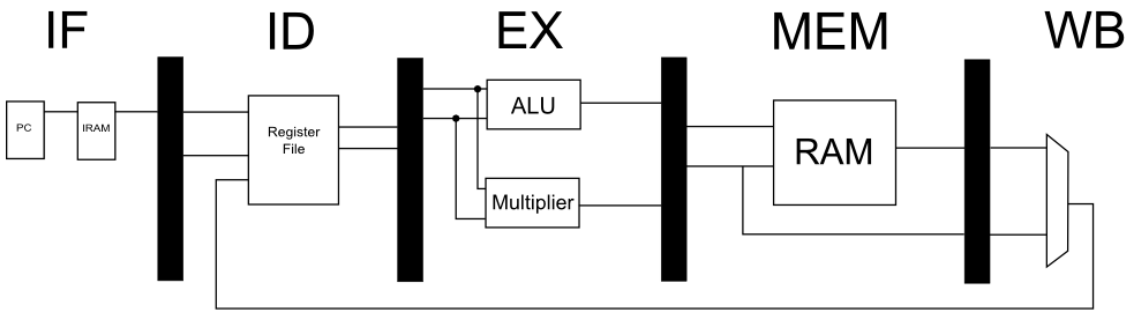

Figure 1: Pipeline

Instruction fetch In this stage there is a register, the *Program Counter* that generate the address for the *IRAM* from which the next instruction is read.

Instruction decode Here the instruction is decoded, the register addresses for the operands are sent to the *Register File* and the *Immediate* field is sign extended.

In order to minimize the branch penalty I choose to put the logic for the computation of the address of jumps in this stage, therefore the when a jump occurs, only one clock cycle is missed. This clock cycle can be filled by the programmer (or the compiler) with another instruction, this is called *Branch Delay Slot*.

Execute This stage contains the *ALU* and the *Multiplier* both of which take the two operands and do the computation.

ALU The *ALU* is made of four different units:

- Adder
- Comparator
- Logic unit
- Shifter

The Adder is implemented using the *Carry Save* architecture and thanks to some external logic it can also work as a subtracter.

The comparator takes as input the difference of the two operands and the selection of the kind of comparison wanted, thanks to some simple logic (like the one seen in class) produce the output.

The logic unit is implemented like the SPARC T2 one.

The shifter is implemented using the embedded VHDL functions, it would be interesting to implement one of the architecture seen in class but those were all implemented at transistor level which were not accessible in our standard cell based workflow.

Multiplier The multiplier is implemented using the *Booth* algorithm as seen in class.

Memory In the Memory stage is implemented the access to the RAM, since it will be not synthesized it is implemented in VHDL in a simple behavioral way.

Write-Back The Write-Back stage selects one data source between RAM output or ALU output and write it into the register file.

2.1 Control Unit

For the *Control Unit* I decided to try something different from the usual, I choose to implement a *Distributed Control Unit* (no special reason, I had it in mind and just want to see if it was better or worse). Instead of generate the entire *Control Word* at the beginning in the ID stage, when the instruction is decoded, and then propagate it through the pipeline I spread it among the five stages and propagate through the pipeline only the instruction. In this way when an instruction reach a certain stage it goes into the *Control Unit* of that stage and this one react (it's a combinational circuit) to it generating the appropriate control signals for the *Data Path*. It is also in charge of hazards handling.

For example when an instruction in the ID stage want to read a register that is written by a previous instruction but this is still in the pipeline and didn't write the register yet, in this case the ID and previous stages (IF) must be stalled until the register is written.

In the end, after working on this style of CU, the more evident difference is that if the instruction coding is not optimal (and the one of DLX were not) the VHDL code is more difficult to manage and maintain.

3 Simulation

In order to simulate the microprocessor the IRAM exploit a VHDL function to load the binary code which contains the instructions that will go out the IRAM. The binary code is generated starting from the assembly code which is assembled by a perl script, this binary code is then further elaborated by another script, conv2memory which simply convert the binary code into an hexdecimal text, this is the format compatible with the VHDL function.

But due to a lot of problem with this workflow I had to bypass the VHDL function and load the code in the IRAM using the modelsim functionality.

4 Synthesis

4.1 Non-optimized

The unconstrained synthesis results are:

- Timing: 1.35 ns
- Power: 4.0373 mW
- Area: 27547 μm^2

The timing analysis is clearly wrong because (I don't know why) it doesn't take into account the multiplier. Moreover there is a problem with the synthesis: design compiler doesn't consider the wire model (I don't know if it's not present in the library or some variable is not set, I generated it from encounter but I wasn't able to set it in design compiler) so the timings, the power and the area are underestimated, this will be a problem in the physical design.

4.2 Optimized

The TCL script for the optimized synthesis is the following one:

```
set TIME_CONSTRAINT 5.0
set POWER_CONSTRAINT 1.0

set_max_dynamic_power $POWER_CONSTRAINT mW
set_max_delay $TIME_CONSTRAINT -from [all_inputs] -to [all_outputs]
create_clock -name "CLOCK" -period $TIME_CONSTRAINT CLOCK
compile -map_effort high -gate_clock -ungroup_all
```

In this way the synthesizer try to achieve as best as it can the listed constraints. I enabled the *Clock Gating* technique in order to save more power, this option automatically find the conditions in which registers are not involved in computation

and disable their clock in order to prevent useless switching which consumes power. I also enabled the ungrouping option in order to remove the hierarchy and let the synthesizer free to optimize also across the limit of the blocks, this should lead in particular to a lower area.

The synthesis results are the following:

- Timing: 4.79 ns (slack = 0)
- Power: 2.0147 mW
- Area: 24159 μm^2

These values are good, I could also relax the constraint on the time and save even more power but this would lead to some problem in physical design which I will explain later.

5 Physical

5.1 Place & Route

For the physical design I first imported the synthesized verilog netlist, then placed the *Power Rings*, the *Power Stripes* and routed them. After this I placed the cells, synthesized a three level clock tree and placed the fillers to fill the placement gap. Then I routed the connection with the two steps routing: *TrialRoute* and *NanoRoute*

5.2 Analysis

In order to perform the post-route analysis I first had to set the operating conditions (1 V and 25 $°C$), then extract the RC parasitics, calculate the delay giving the wanted timing constraints (10 ns to have a clock of 100 MHz). At this point I performed the *Timing Analysis* which generated the two reports.
The *Timing Analysis* reported an arrival time of 12 ns which means a slack violation. Note that there is a difference of 7 ns with respect to the synthesis constraint (this explains why I couldn't set a more relaxed time constraint). My hypothesis is that this big difference is also due to the fact that design compiler doesn't have a wire model for the interconnections, because the wires of the critical path (which is the one that goes through the multiplier) are very long (because the multiplier is very big) and if I try to insert manually a repeater in the middle of the longest wire with the ECO command the timing gets better. As explained before I generated the wire model with encounter but I didn't find a way to import it in design compiler without errors.

5.3 Optimization

At this point I had two options to respect the constraints, the first one was to work at RTL level and break the critical path with a register, but since this path goes

through the multiplier I would had to implement a pipelined multiplier, but this means that I also had to implement in the *Control Unit* the support for multi-cycle instructions. This would be very very interesting but basically endless because of the new kind of hazard that multi-cycle instructions lead to, and since I didn't had that much time the choice must fall back to a faster-to-implement kind of optimization.

This was the *Post-Route Optimization* that achieved its goal and resolved the slack violation.

5.4 Post-Optimization Analysis

Once the slack violation is resolved, power consumption must be analyzed, using as parameters a frequency of 100 MHz and a switching activity of 0.4 the analysis result is 1.5520 mW which meet the constraint.

Other two analysis can also be performed: *IR Drop Analysis* and *Electromigration Analysis*, the result of these two analysis is:

- IRDrop: 5.4672 mV
- Worst Electromigration: 2.5574 mA/u (M8 layer)

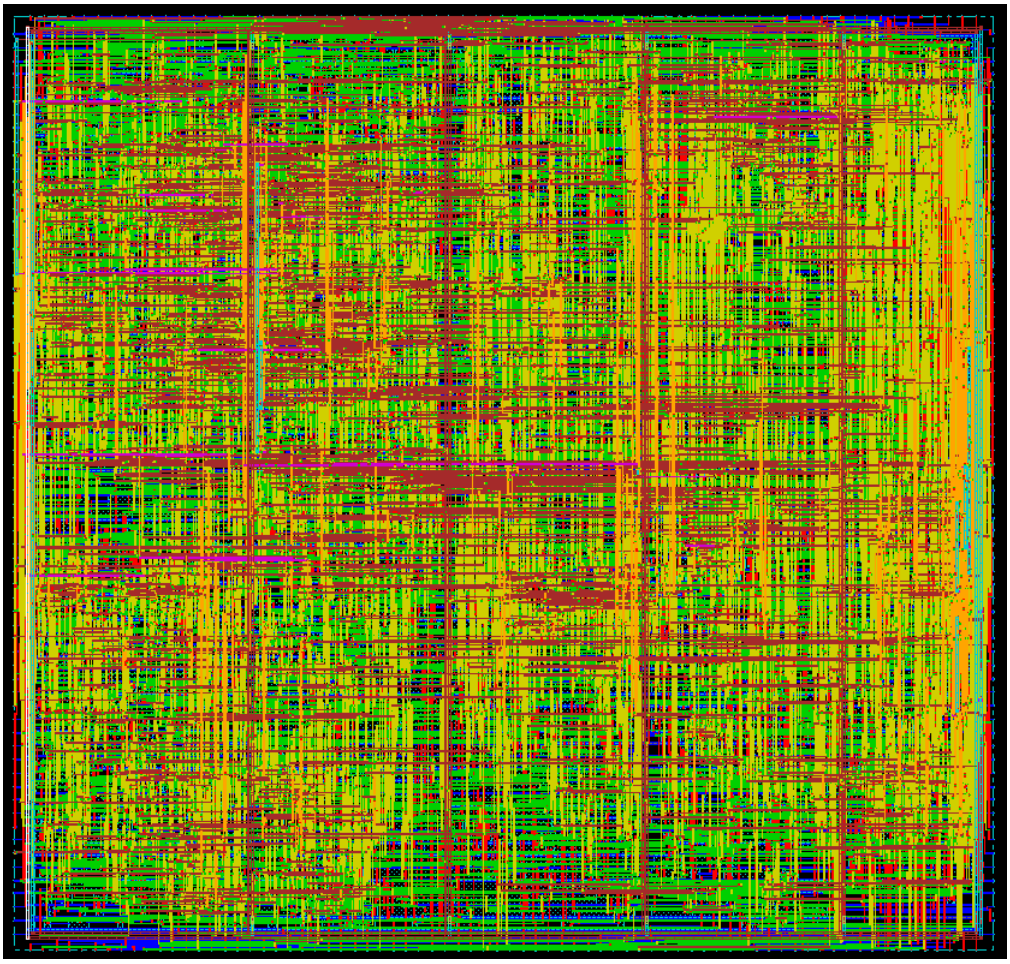

Figure 2: Physical Design