Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: Group16

Augusto Maria Guerriero, Antonio Ras

October 19, 2020

Contents

CHAPTER 1

DLX Features

1.1 Instruction format

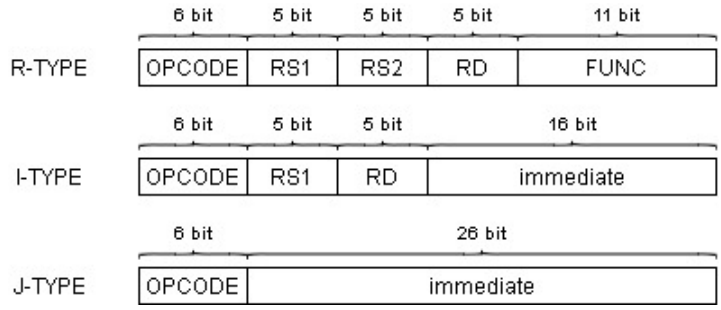


Figure 1.1: DLX instruction formats.

1.2 Supported instruction set

j	jal	beqz	bnez	addi	addui	subi	subui	andi	mult
ori	xori	lhi	jr	jalr	slli	nop	srli	srai	
sequi	snei	slti	sgti	slei	sgei	lb	lh	lw	
lbu	lhu	sb	sh	sw	sltui	sgtui	sleui	sgeui	

Table 1.1: General instructions

sll	srl	sra	add	addu	sub	subu	and	or	xor
seq	sne	slt	sgt	sle	sge	sltu	sgtu	sleu	sgeu

Table 1.2: Register-register instructions

NB: mult instruction, which is normally a floating-point instruction, is operated using integer registers instead: result of a multiplication between two 32-bit integer register is stored on a single 32-bit register, so code must be written with great care to avoid overflows. The assembler has been modified as well to have the mult use the R-TYPE instruction format.

Ex : mult r2 ,r3 , r4
R[regc] <– R[rega] * R[regb]
All are signed integers.

1.3 Key features

Key features of our DLX design are:

1. 5-stage Pipeline
2. Hardwired Control Unit
3. Hazard Detection Unit
4. Dynamic Branch Prediction
5. Data forwarding
6. Multiplier

CHAPTER 2

DLX implementation

2.1 Design choices

As you will see, we often preferred to describe entities with a structural architecture, well aware of the fact that this may not give us the best chance of synthesizing a fast nor power efficient design We thoroughly enjoyed the idea of being responsible for each single logic gate placed in our layout. Having to describe modules in architectural fashion let us understand even more in depth the inner workings of a microprocessor.
The file organization is inspired to the one proposed in the dlx_project_notes; we tried to arrange files in the clearest possible way.
Inside the folder Files/DLX_vhd, home of our DLX implementation for simulation, can be found:

1. DLX_wrapper.vhd, top level entity of our design for simulation: this entity was created to test the DLX microprocessor connected to the memories
2. globals.vhd, file containing *myTypes* package
3. test_bench/ folder, with three
4. DLX_wrapper.core/, folder containing the DLX microprocessor implementation and the code and storage memories.

2.2 myTypes package

All constants used inside the DLX design were grouped into **mytypes** package: here are declared delay values, common sizes and FUNC and OPCODE fields of supported instruction.

2.3 DRAM module

Key feature of the DRAM module we designed in our DLX description are:

1. byte addressing
2. 32-bit data words stored in big-endian format
3. 32-bit input and output data ports

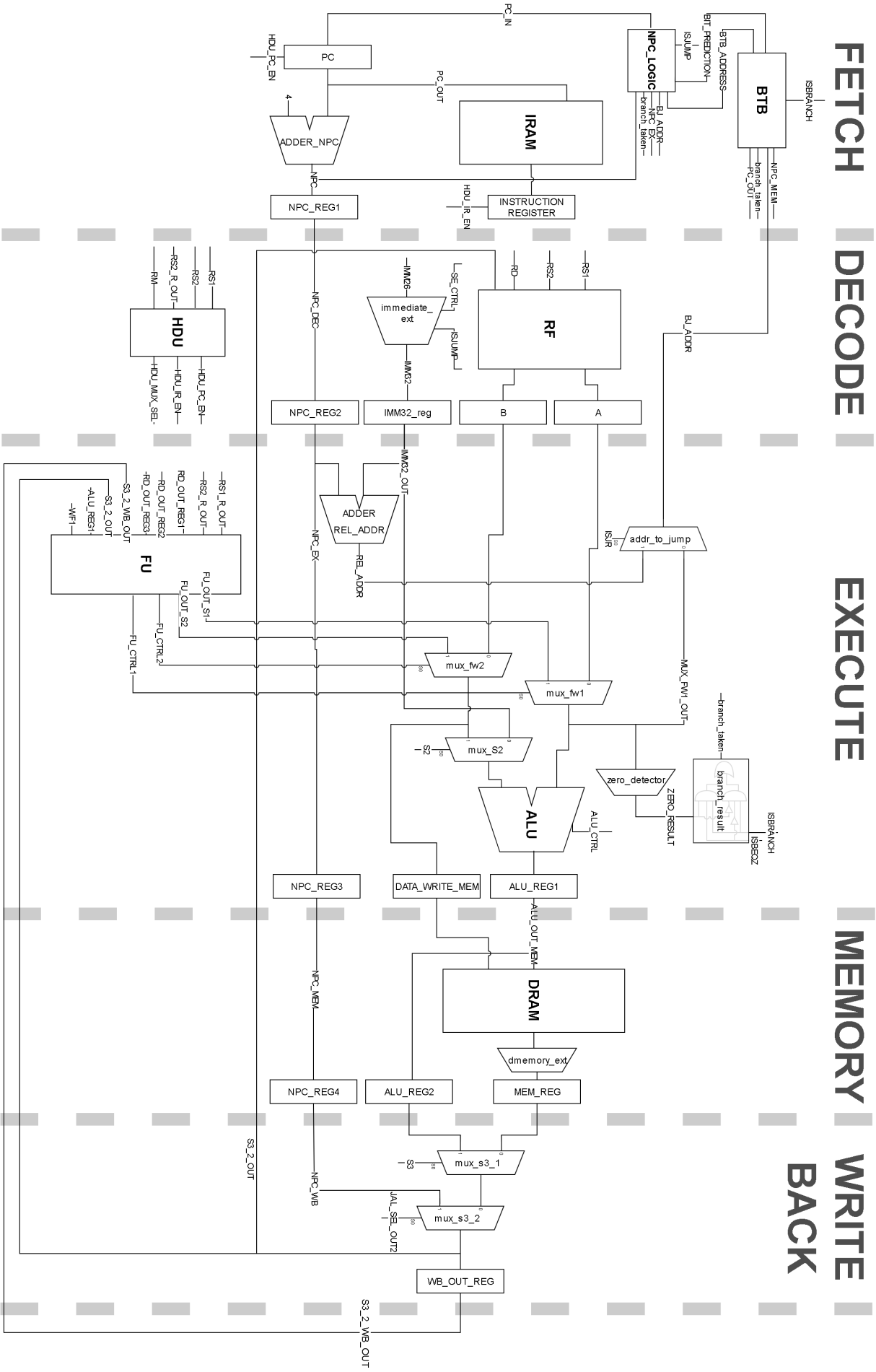
4. one 32-bit address port: at each clock cycle only one read or write operation should be performed
5. 2-bit *size* input signal: the write operation considers the *size* signal to store the desired data (*size*="01" if byte, *size*="10" if half-word, *size*= "11" if word). Read operation always returns a 32-bit data.

2.4 IRAM module

The IRAM module which was provided to us was modified in two ways:

1. the PC is shifted 2 bits right in order to read the correct instruction from the memory
2. the module outputs NOP instructions when End Of File is reached or if reset is asserted.

2.5 Datapath Outline



2.6 Logic Gates

In DLX.core/DATAPATH.core/logic_gates we grouped basic logic gates used for the design of all structurally described modules.
We decided to have as basic logic gates INVERTER and NAND2 ports described behaviourally and have a delay assigned by means of a constant defined in ***myTypes*** package.
Gates XOR2, AND2, OR2, BUFFER are described as structural models using only NAND2s and INVERTERs. BUFFER_GENERIC has also a structural architecture: multiple 1-bit BUFFERs are instantiated with a for-generate statement.

2.7 Blocks

In DLX.core/DATAPATH.core/blocks we grouped basic building blocks.
Structural full adder (FA), behavioural D-flip-flop with asynchronous reset (FD) and structural 2x1 multiplexer(mux21) are used as components for the rest of the modules, which have structural architecture.
The register module (register_generic) is created instantiating multiple flip-flops.
Bigger multiplexers are all structurally described, using the mux21 as building block.

2.8 General Purpose Registers

The DLX architecture include thirty-two 32-bit integer registers; the register_file module was described as a behavioural model.
Our implementation of the RF has two read ports and one write port. Reset and write access are synchronous on the falling edge of the clock. Read access is asynchronous.

2.9 Hardwired Control Unit

In our implementation, the Control Unit is responsible for the generation of control signals for both the Datapath and the ALU.

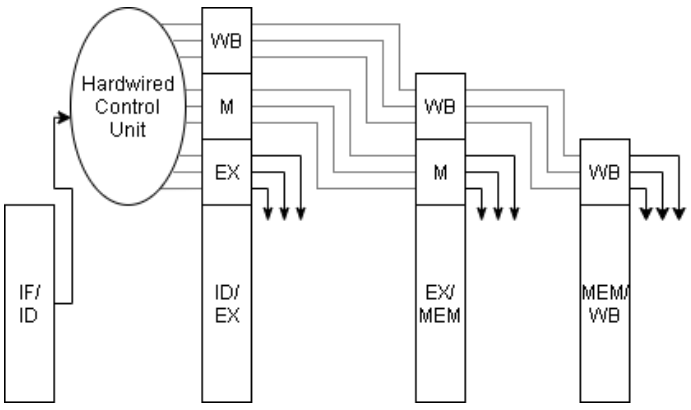


Figure 2.1: Pipelined Control Word schematic.

The control unit rely on a 20-bits wide, 64-words deep ROM, accessed using the OPCODE (6-bits wide) of the fetched IR as address. Each word in memory contains a Control Word which consists of

20 control signals, 3 of which are used in the ID stage, 8 in the EX stage, 6 in the MEM stage and 3 in the WB stage.
The ROM is accessed with a dataflow statement.
Alongside the signals in the CWs in memory, we also have signals SE_CTRL and ALU_CTRL: both signals cannot be put inside the ROM as in the case of an R-type instruction, which has always Opcode "000000", they change also in relation to the FUNC field of the IR.
We use two behavioural processes:

1. CW_PIPE: it ensures that the CW and ALU_CTRL are sent to the Datapath in a synchronous fashion. Signal SE_CTRL is placed inside the CW to send out using concatenation.

2. ALU_SE_CTRL: has the task to generate correct control signals for ALU and sign-extension units decoding both Opcode and FUNC.

The CW is generated at the Decode stage and is then passed through a set of pipelined register which will ensure the correct timing and hold time for all subsequent stages.

2.10 Data Forwarding Unit

We decided to implement a Forwarding to limit the performance deficit derived from pipeline stalls resulting from data hazards.
Our Forwarding Unit is programmed to forward to the current EX stage, data coming from the EXE stage of the previous instruction, from the MEM stage of two instruction before and the WB of three instruction before. This way RAW stalls are almost completely avoided.

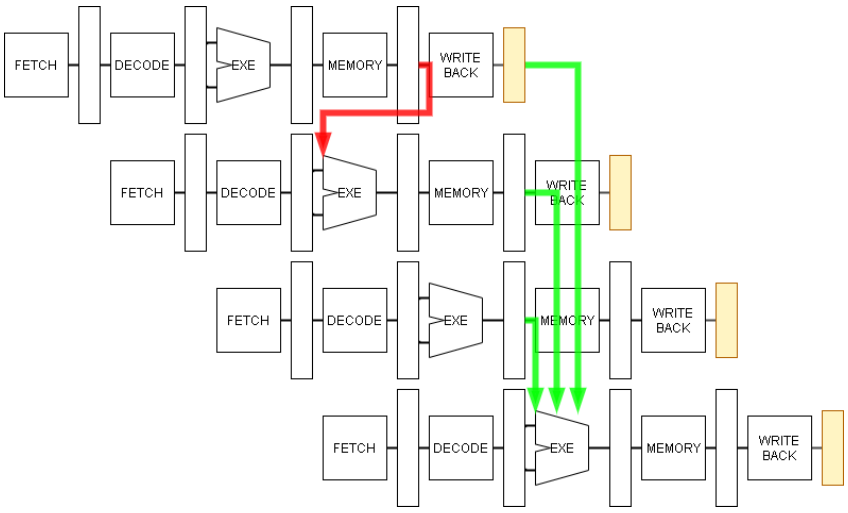


Figure 2.2: In green Data Forwarding. In red a load instruction which causes a not solvable Data Hazard.

Placed in the Decode stage, the Forwarding Unit checks values of RS1 and RS2 (operand registers from IR) against the RDs (destination register) coming from EXE, MEM and WB stage. If a match is found, the FU will forward the required value to the multiplexer placed at the correct input of the ALU and will send control signals to the said multiplexer.
Highlighted in orange, the registers we put after the Writeback stage to pipeline both RD and the result written to the Register File: this ensures that the timing of write and read accesses to the register file does not cause any problems.

2.11 Hazard Detection Unit

As it was anticipated, the Data Forwarding module eliminates almost all Data Hazards, but there is still one case that in which a Data Hazard occur.

```
lw r3, 0(r2)
addi r3, r3, 10
```

If an instruction has as operand a register whose value is loaded from memory in the previous instruction, the correct value is not available in the EXE stage of the latter instruction.
When this happens, the pipeline is stalled for one clock cycle to resolve the Hazard.
In a pipeline stall, fetch and decode stage are halted so values of the not yet executed instructions are preserved till the value read from memory is available; the stall causes a "bubble" in the pipeline, delaying all instructions after it.

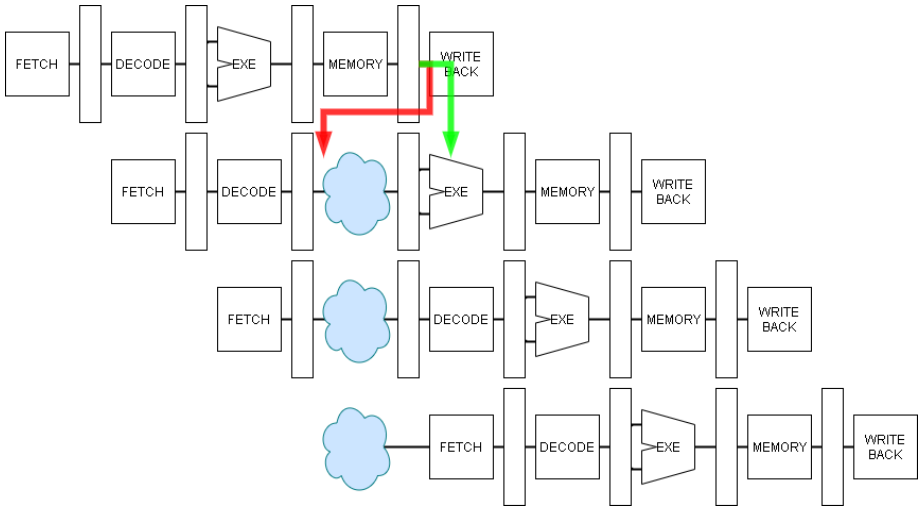


Figure 2.3: Pipeline stall and pipeline "bubble" caused by a Data Hazard.

Our HDU, placed in the decode stage, receives as input RS1, RS2, operands of the current instruction, RD_EX and MEMRD_EX, respectively destination register and 1-bit DRAM read signal of the previous instruction. When conditions are met, being one of the operands is the destination register of an immediately preceding load instruction, PC and IR registers are disabled and the Control Word is temporarily reset.

2.12 Branch Table Buffer

We decided to include an 8-entry Target Buffer with 2-bit Saturation Prediction State.
During the fetch stage, in the BTB block, a check is performed to see if the PC register is stored in memory and if its value is valid, then the prediction state will be sent to the Datapath.

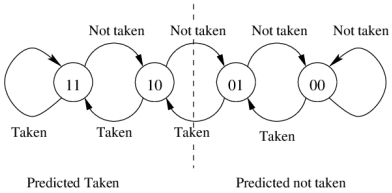


Figure 2.4: 2-bit Prediction State, saturating counter.

Bits PC[4:2] of the Program Counter are used to access the BTB (the two least significant bits are ignored as they are always '0', being the address always a multiple of 4). If the current address data is valid and PC is a match, the Prediction bits are evaluated and a prediction is sent to the Datapath, along with the target in the case of a "taken" prediction.

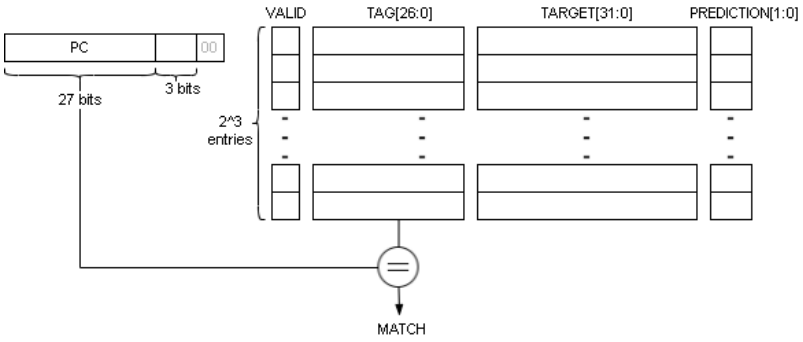


Figure 2.5: 32-bit 8-entry Branch Table Buffer with 2-bit Prediction State.

Prediction bits are updated following the saturating counter model, shown in figure 2.4.
The method we implemented is shown to greatly improve performance compared to a static branch prediction, and is proven to be a good choice even among other techniques. In the paper from Scott McFarling "Combining Branch Predictors", the SPEC'89 benchmarks report that very large bimodal predictors saturate at 93.5% correct, once every branch maps to a unique counter. Obviously conditions differ a lot with our case, but studies like this prove the quality of the idea.

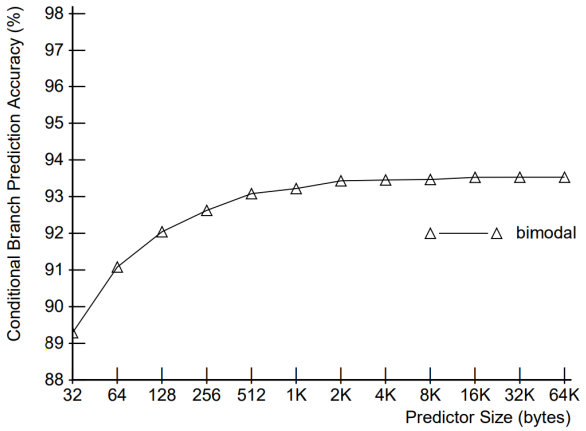


Figure 2.6: Bimodal Branch Predictor performace.[McFarling, Scott (June 1993). "Combining Branch Predictors"]

2.13 ALU

The top entity of the ALU is a wrapper which connects components ALU_control and ALU_datapath. ALU_control was considered necessary as the ALU_ctrl signal received from the Control Unit must be translated into more specific signals needed inside the ALU_datapath.
ALU_datapath consists of four blocks: a module which performs additions, subtractions and comparations, a shifter, a multiplier, and a logic unit.

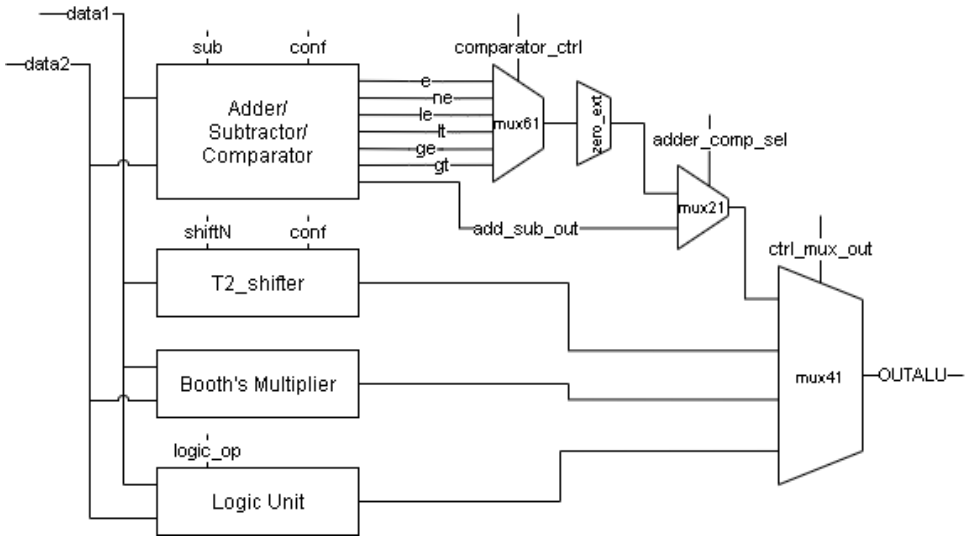


Figure 2.7: ALU_datapath schematic.

2.13.1 Adder/Subtractor/Comparator

The add_sub_comp_block's main component is without a doubt the adder, in our case designed with the P4_adder architecture.
The P4_adder is used to perform addition and subtraction and its output signals are then fed to a logic net which generates the output bits for the comparator.

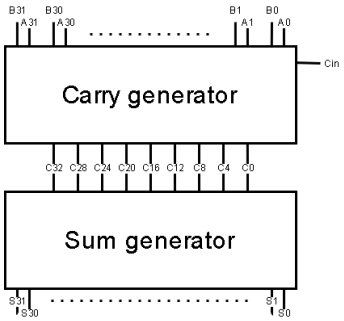


Figure 2.8: 32-bit P4 adder top-level entity schematic with 4-bit carry-select sum generators.

2.13.1.1 P4 Adder

For the adder module, not just within the ALU, we created a generic structural design inspired to the Pentium 4 adder.
The top level entity of the P4 adder connects signals from entities carry_generator and sum_generator.
Sum Generator
The sum generator receives precomputed carry values from the carry generator module and produce the sum on output.
In the structural architecture, a series of carry-select sum generator modules (CSblock) is instantiated with a for-generate concurrent statement: the number for CSblock to be instantiated depends on values of the NBIT variable and the NBIT_PER_BLOCK constant (32 and 4 respectively in the DLX implementation).

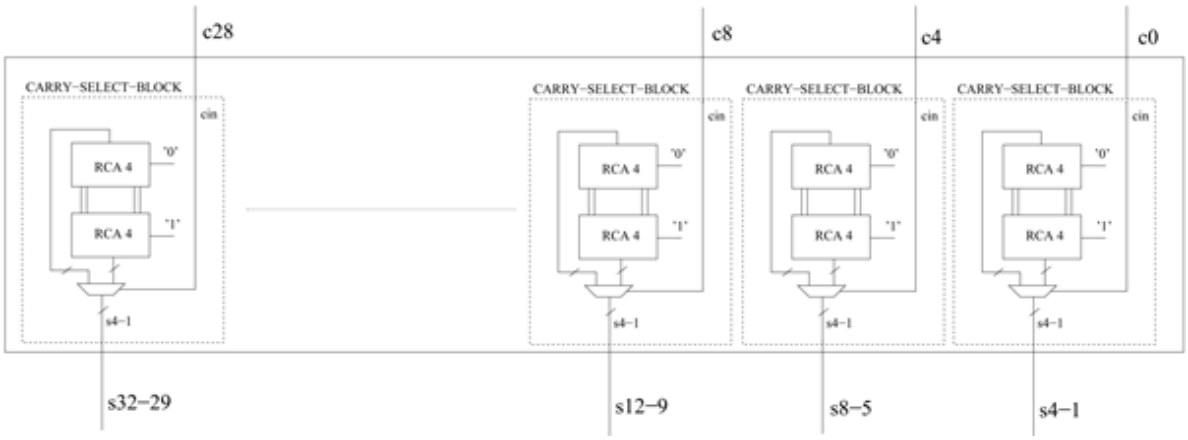


Figure 2.9: Sum Generator schematic. ["Microelectronic Systems Lecture Notes", Mariagrazia Graziano, 2020, p.99]

The goal of CSblock is to produce two sums in parallel: one assuming the carry-in is '1', the other assuming it is '0'. The correct output is selected with a multiplexer controlled by the carry-in.
Inside the carry-select sum generator architecture, two Ripple Carry Adders are declared, plus a generic 2x1 multiplexer controlled by the carry received from the Carry generator module. The size of the RCA and the multiplexer are dictated by the NBIT_PER_BLOCK constant.
The generic Ripple Carry Adder has a structural architecture and is composed of a series of full adders.
Carry Generator The goal of this module is to pre-compute all the carries, starting from the two operands without knowing in advance the actual partial sum.
The algorithm is based on two concepts: propagation and generation. A generation happens when a carry bit is generated by the sum of two bits; a propagation happens when one of two bits are 1 so that a carry can propagates through. The general generate of a stage j at level k+1 is equal to 1 if there is a generation of a carry in j or if there is a propagation in j and a generation in i at level j

$$G_{i:j} = G_{i:k} + P_{i:k} * G_{k-1:j}$$

The general propagate of a stage j at level l+ is equal to 1 if there is a propagate both in stage j and i at level l

$$P_{i:k} = P_{i:k} * P_{k-1:k}$$

We implemented this module with a structural architecture and as the implementation is pretty complex we decided to produce a well commented code to avoid cluttering the report (at DLX_wrapper.core/DLX.core/ALU.core/add_sub_comp_block/add_sub_block/P4_adder/carry_generator.vhd).

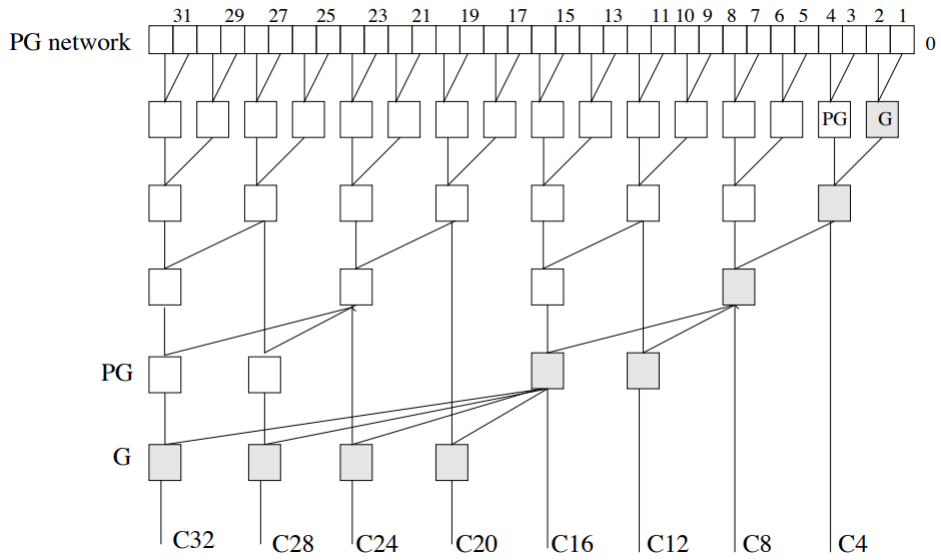


Figure 2.10: Sparse Tree Carry Look-ahead schematic. ["Microelectronic Systems Lecture Notes", Mariagrazia Graziano, 2020, p.106]

2.13.1.2 Subtractor

The second operand of the adder, B in figure, is bitwise XORed with the control signal SUB, which also is connected to the Cin input of the adder: when SUB='0', a normal addition is performed, when SUB='1', B is bitwise complemented and, because Cin ='1', the resulting operation is SUM=A+B'+1, which corresponds to SUM=A-B in two's complement.

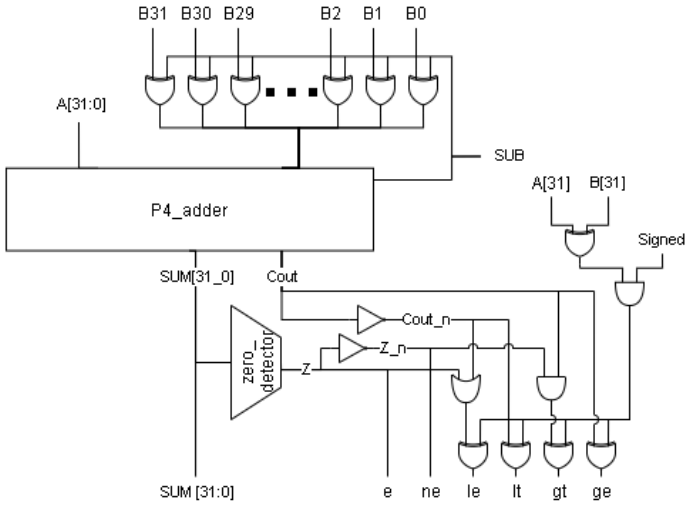


Figure 2.11: add_sub_comp_block schematic.

2.13.1.3 Comparator

If the requested operation is a comparison, the add_sub_block must perform a subtraction, thus SUB='1'; the outputs of an unsigned comparator are then obtained as follows:

A > B ⇒ C and Z'
A ≥ B ⇒ C
A < B ⇒ C'
A ≤ B ⇒ C' or Z
A = B ⇒ Z
A /= B ⇒ Z'
The comparator was modified to be valid also for comparison between signed operands. We noticed that outputs *le*, *lt*, *ge*, *gt*, were negated when the comparator had to consider the operands as signed and the operands had opposite sign: signals *le*, *lt*, *ge*, *gt* were negated in this case.

2.13.2 Booth's algorithm Multiplier

We decided to use a multiplier based on Booth's algorithm which has the advantage of generating the product without really executing a product, but simply additions and left shifts, as only powers of 2 are used. The algorithm is based on pre-shifted values, computed starting from the first operand. Multiple 3to3 encoders, whose input is the second operand, select the values to sum. We chose the P4 adder to operate the sums, having the goal to reach a tradeoff between area and performance. Although it has an high parallelism, its performance is limited by the delay due to the sum of halves; the partial shifted values can be obtained in parallel and easily, as a simple left shift takes place.

2.13.3 T2 Shifter

We decided to design our shifter following the architecture of the T2 Niagara processor.
The T2 shifter operates the shift in one clock cycle and it is organized in three different levels: in this report the 32-bit implementation of the T2_shifter. will be discussed.
The first level produces masks in which a coarse grain shift is performed ; the second level selects the mask whose coarse grain shift is closer to the one required; the third level performs the fine grain shift and the result is presented to the output.
The first level produces eight 39-bit different masks (32-bit data +7), four for the left shift, four for the right shift. The first masks, ML0 and MR0 (mask0 for left and right shift) are produced with a dataflow statement. The remaining masks, (ML8, MR8, ML16, MR16, ML24, MR24) are generated with a concurrent for-generate statement which also instantiates one 2in1multiplexer for each set of left and right shift masks. $conf(0)$ is used to choose the masks with the shift performed in the required direction.

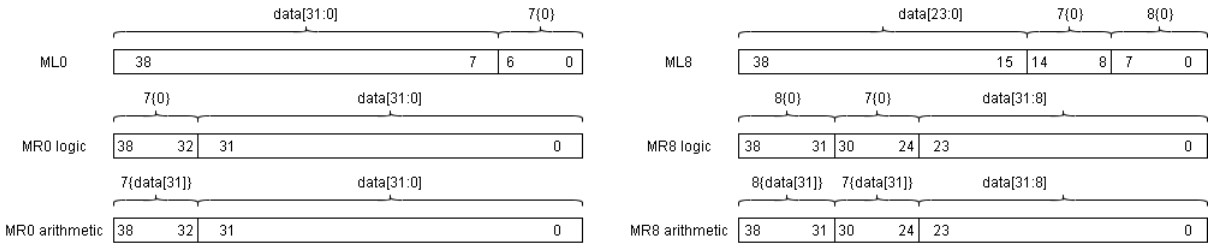


Figure 2.12: 32-bit T2 shifter, masks M0 and M8.

In the second level the four selected masks are then fed into a 4in1 39bit multiplexer: the output is selected using the two most significant bits of the *shift* signal.
Lastly, in the third level, fine grain shift is performed using an 8in1 32-bit multiplexer: the 39-bit mask obtained in stage 2 is create eight 32-bit masks with progressive 1-bit shift. This multiplexer is controlled using the 3 least significant bits of the *shift* signal, which are negated in case of a left shift.

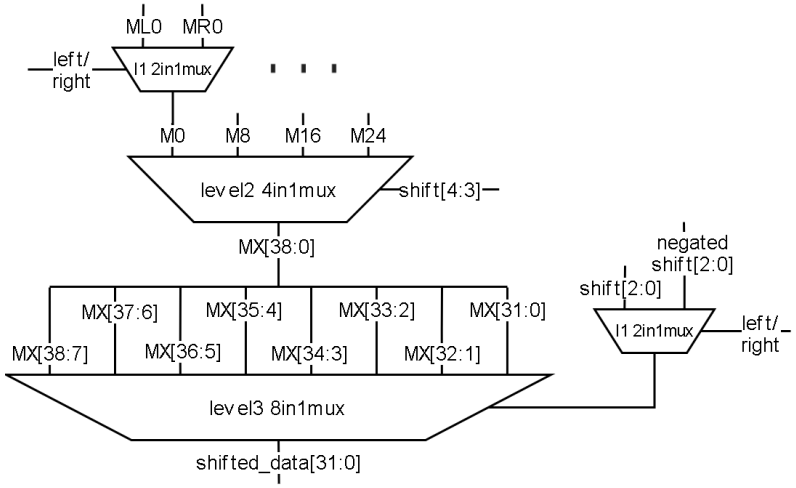


Figure 2.13: 32-bit T2 shifter schematic.

2.13.4 T2 Logic Unit

Even if the T2 Logic Unit is knowingly excessive for our needs, as the DLX Datapath is only required to perform AND, OR, XOR operations, we decided to design and implement a module inspired to it. The 1-bit logic block shown in Figure 2.14, consisting of four NAND3 and one NAND4 is multiply instantiated by a for-generate statement in our Logic Unit.
As it can be seen in the table to the right side, for our purposes, signal S0 is always 0, thus L0(NAND3) was deleted and the NAND4 has been replaced by a NAND3, as this does not change the truth table for our purposes.

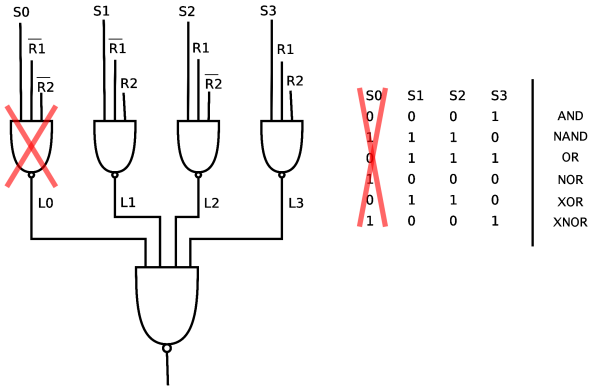


Figure 2.14: T2 Openspark 1-bit logic block and relative control signals table. Crossed in red, parts eliminated in our design.

CHAPTER 3

DLX Synthesis

3.1 Timing performance

We initially tried to synthesize with a very short clock cycle, 2ns, and a coherent max_delay constraint. The obtained worst delay was unsurprisingly much higher than 2ns, at 9.64ns: this can be explained by the fact that we choose not to use a high-effort compile command in addition to our mostly-structural design, which leaves less freedom to the synthesizer.
We synthesized again with a clock of 10ns and this time DESIGN_VISION was able to meet the constraints, with a resulting max delay of 9.96ns, barely inside the time window of a clock cycle minus the setup time. The reason for the delay being higher than the first synthesizer is probably due to the fact that the synthesizer "eats" all the available positive slack to balance the overall area/power/timing cost.
Looking at the timing report, it is clear that the critical path is the one that traverses the multiplier block, whose delay amounts to more than 90% of the total path; this was expected, as the multiplier is the most complex unit in our DLX design.

```
Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
DLX                5K_hvratio_1_1        NangateOpenCellLibrary

Point                                                   Incr       Path
--------------------------------------------------------------------------
DP/RD_reg2/ffd2_0/Q_reg/CK (DFFR_X1)                    0.00       0.00 r
...
DP/alu_block/datapath/MULT/B_mp[2] (boothmul_NBIT32)    0.00       0.70 r
...
DP/alu_block/datapath/MULT/Y_mp[31] (boothmul_NBIT32)   0.00       9.79 f
...
DP/alu_reg1/ffd2_31/Q_reg/CK (DFFR_X1)                  0.00      10.00 r
library setup time                                     -0.04       9.96
data required time                                                 9.96
--------------------------------------------------------------------------
data required time                                                 9.96
data arrival time                                                 -9.96
--------------------------------------------------------------------------
```

```
  slack (MET)                                                              0.00
```

3.2 Power performance

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      1.2803         352.9159           71.7659           354.2680  (   9.60%)
register       1.4822e+03           4.9624        1.6367e+05        1.6509e+03  (  44.73%)
sequential     4.9329e-02        7.5724e-03        4.9405e+03           4.9974  (   0.14%)
combinational     38.0795          72.1862        1.5706e+06        1.6809e+03  (  45.54%)
--------------------------------------------------------------------------------------------------
Total          1.5216e+03 uW       430.0721 uW     1.7393e+06 nW     3.6910e+03 uW
```

3.3 Area occupation

The units are in square micron. The cell area can sometimes be different from the total area as the total cell area includes just the standard cells, while the total area can include the Net interconnect area as well. In this case the Net interconnect area is not available, thus the total area is undefined.

```
Number of ports:                          167
Number of nets:                           210
Number of cells:                            2
Number of combinational cells:              0
Number of sequential cells:                 0
Number of macros:                           0
Number of buf/inv:                          0
Number of references:                       2

Combinational area:       67106.745088
Noncombinational area:    10430.923856
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          77537.668945
Total area:                 undefined
```

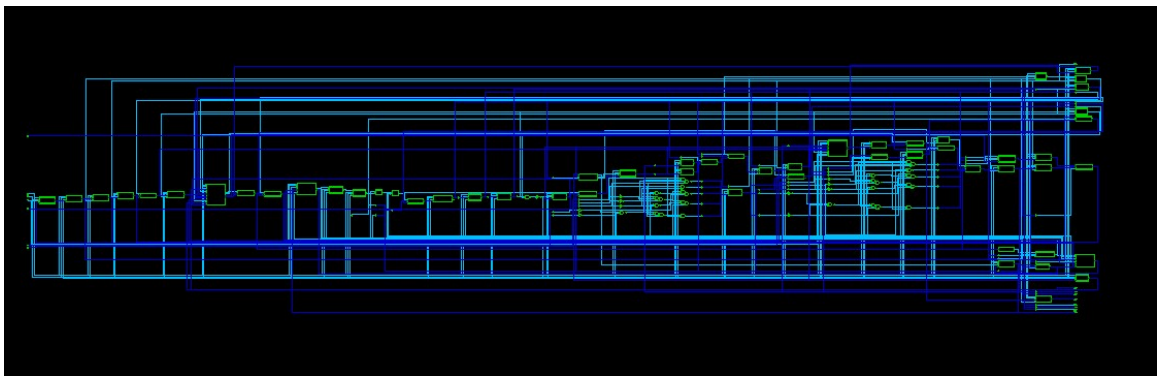


Figure 3.1: Datapath architecture, post-synthesis schematic.

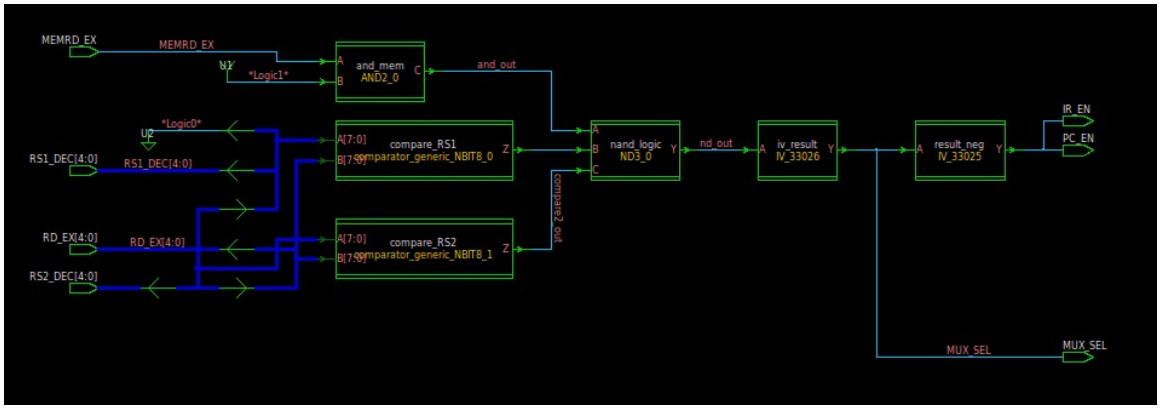


Figure 3.2: Hazard Detection Unit architecture, post-synthesis schematic.

CHAPTER 4

DLX Phisical Design

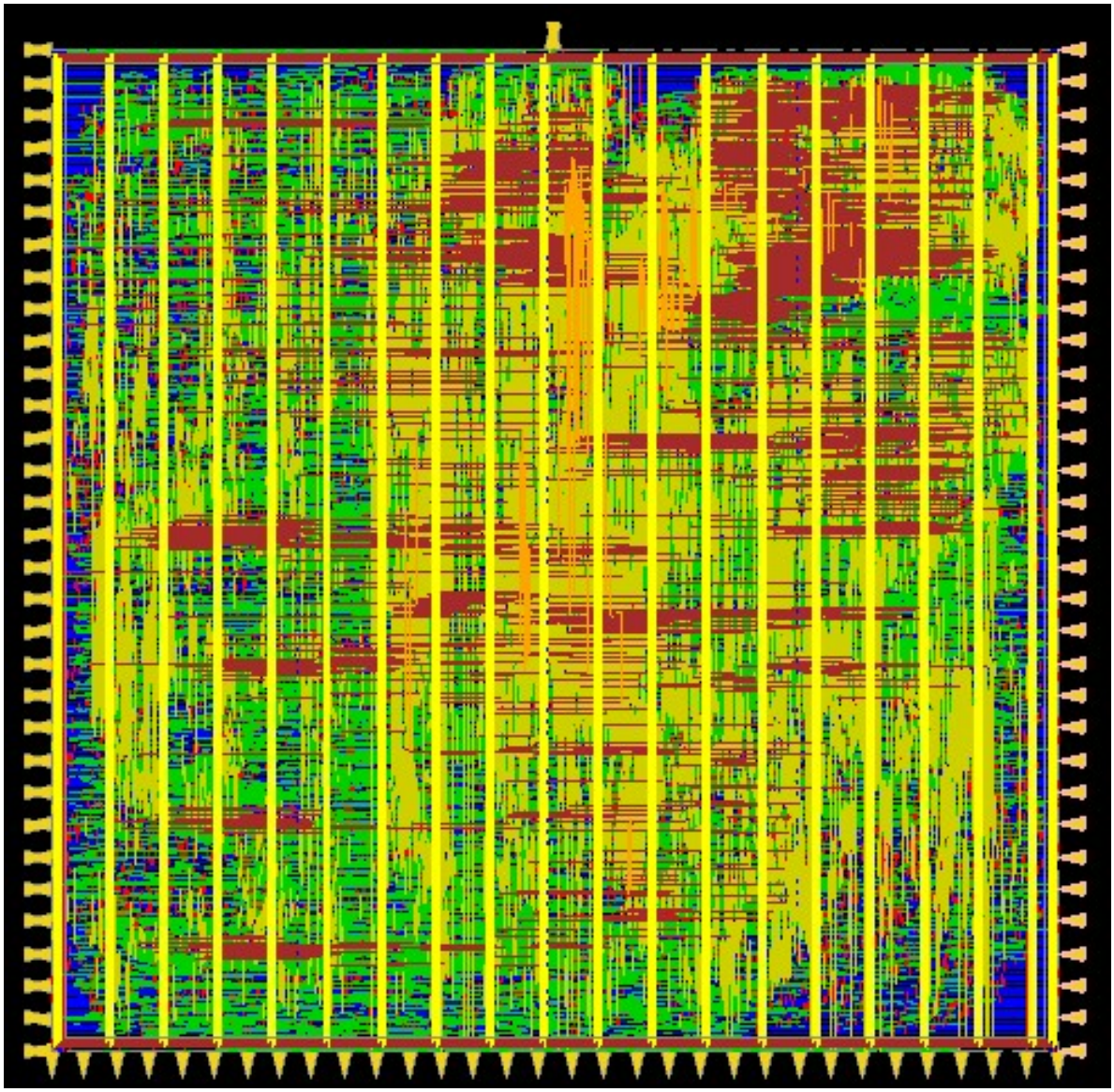


Figure 4.1: Innovus screen dump of placed, routed and optimized DLX.