Politecnico di Torino

Microelectronic Systems

Student: Luca Genca
ID: s345156

Student: Alessio Delli Colli
ID: s345701

December 3, 2025

1 Intruduction

The project consists in the creation of a dlx processor in vhdl. The dlx is a RISC processor with a Harvard architecture, which means that it uses two different memories for data and instructions. The processor is organized in a five-stage pipeline. At instruction fetch (IF) the control unit reads the current instruction pointed to by the PC; decode (DE) reads the register file and prepares operands and immediates; execute (EX) performs ALU operations; memory (MEM) performs data memory accesses; and writeback (WB) updates the register file.
It has 3 different types of instructions:

Instruction Type	**Description and Example**
R-type	Instructions that operate on two source registers and store the result in a destination register. Example: `add $r1, $r2, $r3`
I-type	Instructions that involve a register and an immediate value. Used for arithmetic with constants, memory access, and branching. Example: `addi $r1, $r2, 10`
J-type	Instructions perform unconditional jumps. Example: `j 40`

Table 1: DLX Processor Instruction Types

The complete implemented instruction set is the following table:

1.1 Pro features

The pro features we have implemented are:

- extended instruction set (including division)
- branch prediction unit
- Hazard control with the use of stalls

Table 2: Implemented Instructions with Opcodes and Function Codes

I/J-type	**Opcode**	**Int R-type**	**Func**	**Float R-type**	**Func**
j	0x02	sll	0x02	multf	0x02
jal	0x03	srl	0x03	divf	0x03
beqz	0x04	sra	0x07		
bnez	0x05	add	0x20		
addi	0x08	addu	0x21		
addui	0x09	sub	0x22		
subi	0x0a	subu	0x23		
subui	0x0b	and	0x24		
andi	0x0c	or	0x25		
ori	0x0d	xor	0x26		
xori	0x0e	seq	0x28		
lhi	0x0f	sne	0x29		
jr	0x12	slt	0x2a		
jalr	0x13	sgt	0x2b		
slli	0x14	sle	0x2c		
nop	0x15	sge	0x2d		
srli	0x16	mult	0x0e		
srai	0x17	div	0x0f		
seqi	0x18	sltu	0x3a		
snei	0x19	sgtu	0x3b		
slti	0x1a	sgeu	0x3d		
sgti	0x1b				
slei	0x1c				
sgei	0x1d				
lb	0x20				
lbu	0x24				
lhu	0x25				
lw	0x23				
sb	0x28				
sw	0x2b				
sltui	0x3a				
sgtui	0x3b				
sleui	0x3c				
sgeui	0x3d				

2 Control Unit

We used a hardwired control unit with a 23-bits control word

Table 3: Control signals and their meanings

Signal	Meaning
PC_LATCH_EN	Enables updating of the program counter (PC) register.
RegA_LATCH_EN	Enables the register containing the first operand in I and R type instructions.
RegB_LATCH_EN	Enables the register containing the second operand in R type instructions.
RegIMM_LATCH_EN	Enables the register containing the immediate value in I and J type instructions..
RFR1_EN	Enables read signal for register-file read port 1 (reads operand A).
RFR2_EN	Enables read signal for register-file read port 2 (reads operand B).
RF_EN	Global register-file enable.
ALU_OUTREG_EN	Enables saving of the ALU result into the ALU output register for the next stage.
MUX_B	Selects signal for the ALU second operand multiplexer (e.g. choose between RegB / immediate).
MUX_A	Selects signal for the ALU A second operand multiplexer (e.g. choose between RegA / PC).
MEM_LATCH_EN	Enables saving of address/data into the memory-stage pipeline register.
EQ_COND	Equality condition (used for branching).
JUMP_EN	Enables jumping (true for both jumps and branches).
JUMP	Indicates that a jump needs to be performed instead of a branch.
BYTE	Select byte-sized memory access (8-bit load/store).
DRAM_WE	DRAM write enable.
LMD_LATCH_EN	Enables the memory register.
SEL_MEM_ALU	Select between memory data and ALU result for the write-back.
RF_WE	Register-file write enable.
JAL	needed for the JAL instruction.
HALF_WORD	Select half-word memory access (16-bit load/store).
H_L	High/Low half/byte selector (chooses high-half vs low-half or upper vs lower byte when writing).
S_U	Sign/Unsigned control for load and store: chooses between sign-extension (signed) or zero-extension (unsigned).

3 Datapath

The datapath forms the structural backbone of the processor. Its components are:

- **Branch Target Buffer (BTB):** saves recently encountered branches and respective targets, enabling the program counter to jump immediately when seeing a branch or jump, thus reducing stalls.
- **Register File:** an array of registers with two asynchronous read ports and one synchronous write port.

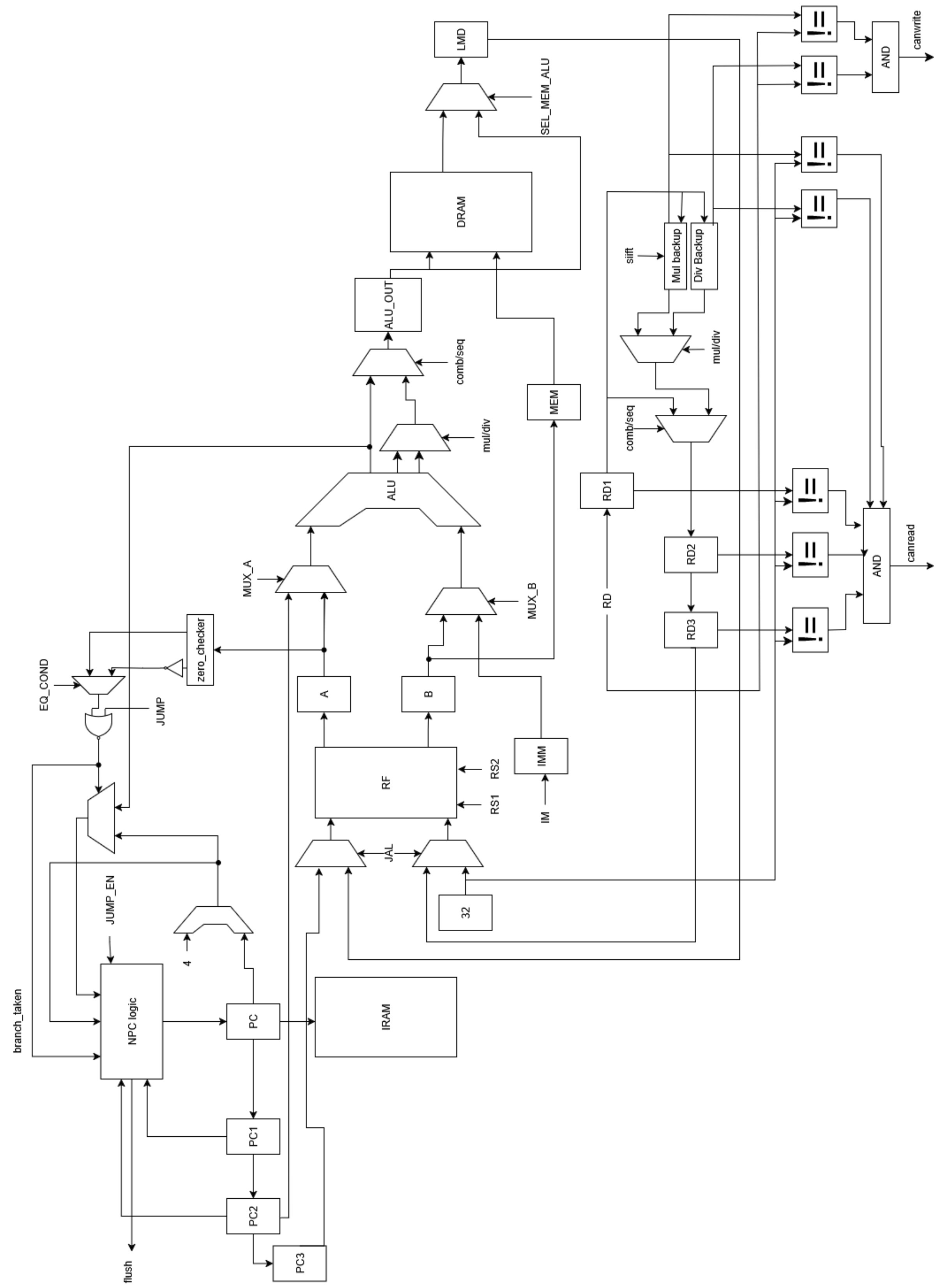
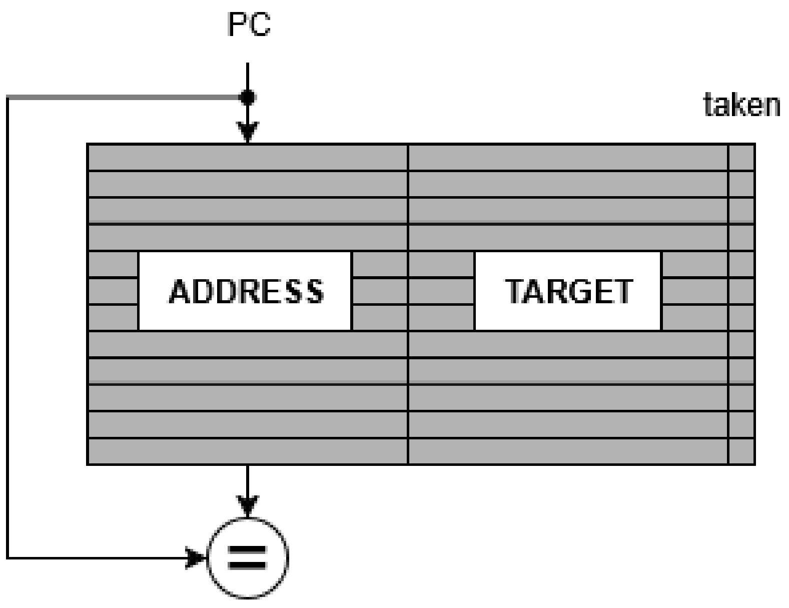


Figure 1: schematic of the entire datapath

- **Arithmetic Logic Unit (ALU):** performs integer arithmetic, bitwise logic, shift, comparison operations, multiplications, and divisions.
- **Pipeline / Skew Registers:** small registers inserted between pipeline stages or to skew signals and align timing. These registers are also used for hazard detection.

3.1 Branch Target Buffer (BTB)



The BTB follows a simple prediction algorithm based on the last observed behavior of each branch. Whenever a branch or jump instruction is executed, the BTB entry corresponding to its program counter (PC) is updated with the new outcome — either *taken* or *not taken*. On subsequent encounters of the same instruction, the processor consults the BTB and uses the most recent prediction to decide whether to fetch from the branch target address or continue sequentially. This one-bit prediction strategy may mispredict alternating branches, but it performs well for branches with stable behavior, such as loops.

3.2 ALU

The ALU is composed of several components: adder, comparator,logic unit, pipelined multiplier, shifter and divider.

3.2.1 Adder

The adder used in the ALU is based on the Pentium 4 adder architecture.

It is composed of two main parts:

1. a sparse carry-generation tree, which computes the carries using group propagate and generate signals;

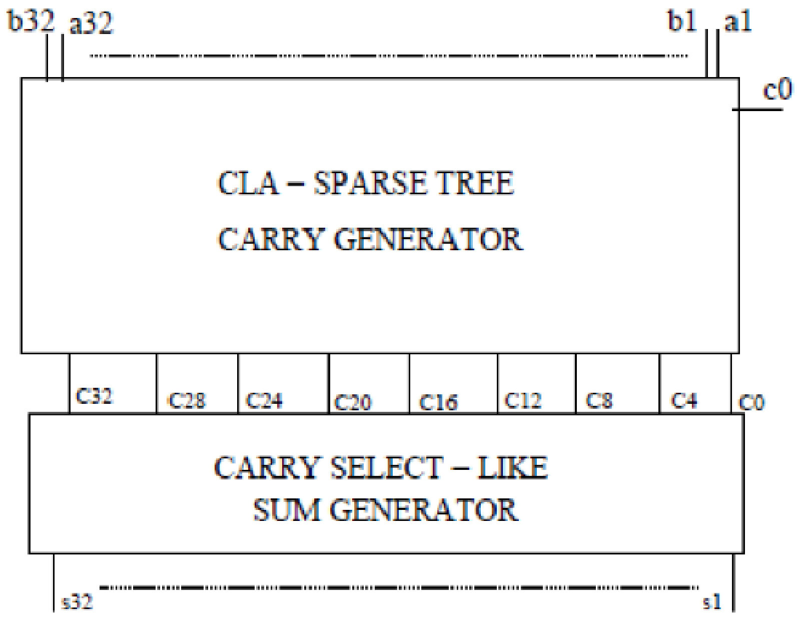


Figure 2: Pentium 4 adder structure

2. a carry-select stage, which uses the carries from the sparse tree to select the correct partial sums.

The sparse tree makes use of two types of blocks, **PG** and **G**, whose outputs are computed recursively as:

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

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

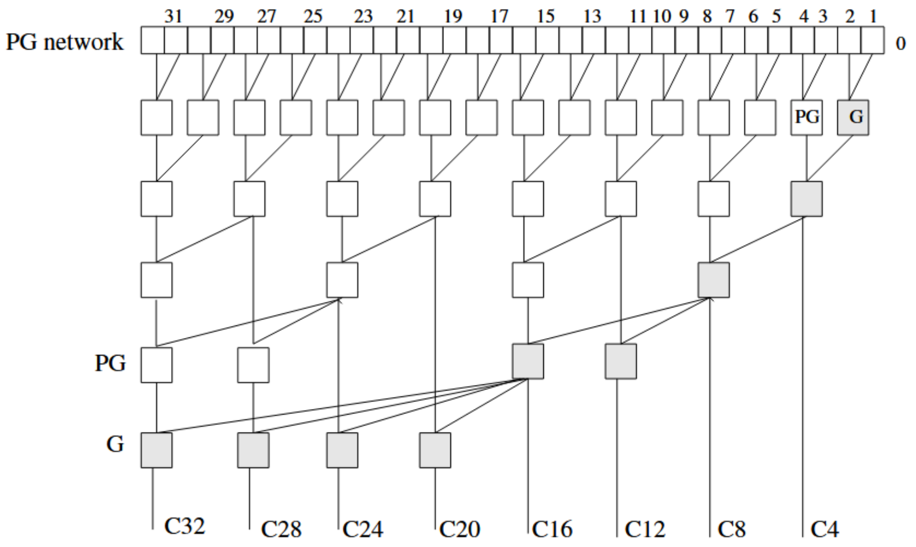


Figure 3: Structure of the propagate/generate network

The carry-select block consists of several small ripple-carry adders (RCAs). Each RCA computes the partial sum assuming both possible carry-in values (0 and 1).A multiplexer then selects the correct partial sum based on the carry provided by the sparse tree.

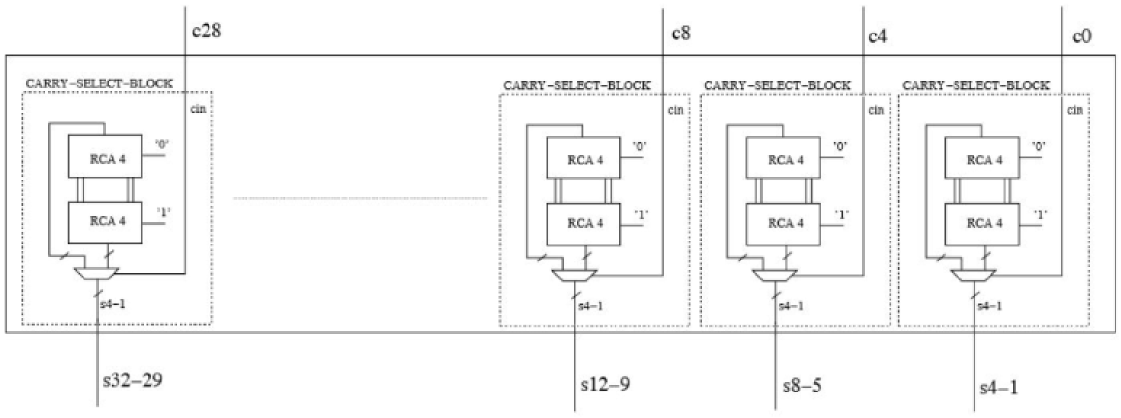
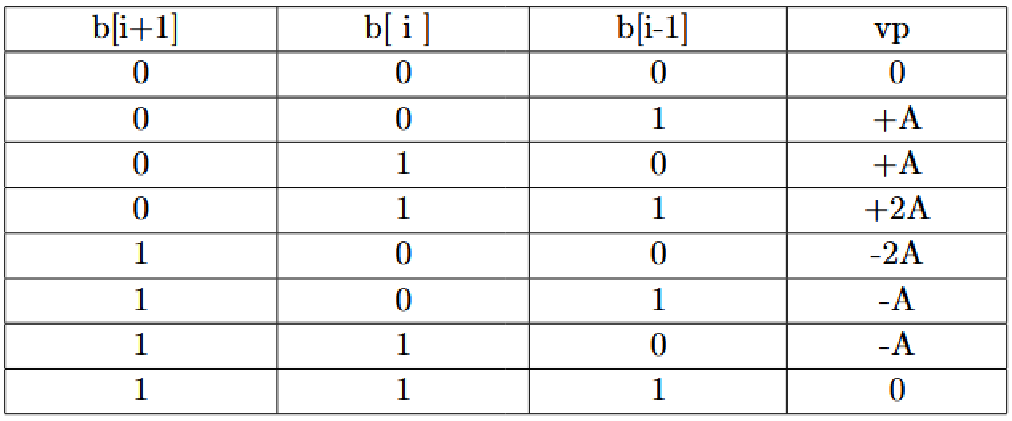


Figure 4: Carry-select block

3.2.2 Multiplier

The multiplier is based on the Booth algorithm. The second operand is partitioned into groups, resulting in $N/2$ groups for an N-bit operand. Each group is processed by a Booth encoder, which determines the corresponding operation (e.g., 0, $\pm A$, or $\pm 2A$). The partial products selected by the encoders are then forwarded to a pipeline of adders.



b[i+1]	b[i]	b[i-1]	vp
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

Figure 5: Booth encoder

These adders accumulate the partial results, with multiplexers selecting the appropriate input based on the encoder outputs. Pipeline registers are inserted between stages to reduce the critical path and allow the multiplier to operate at a higher clock frequency.

3.2.3 comparator

The comparator computes all signed and unsigned comparison conditions required by the DLX. It receives as inputs the operands A and B, the sum produced by the ALU adder $(A - B)$, and the carry-out bit. From these signals, the unit derives both signed and

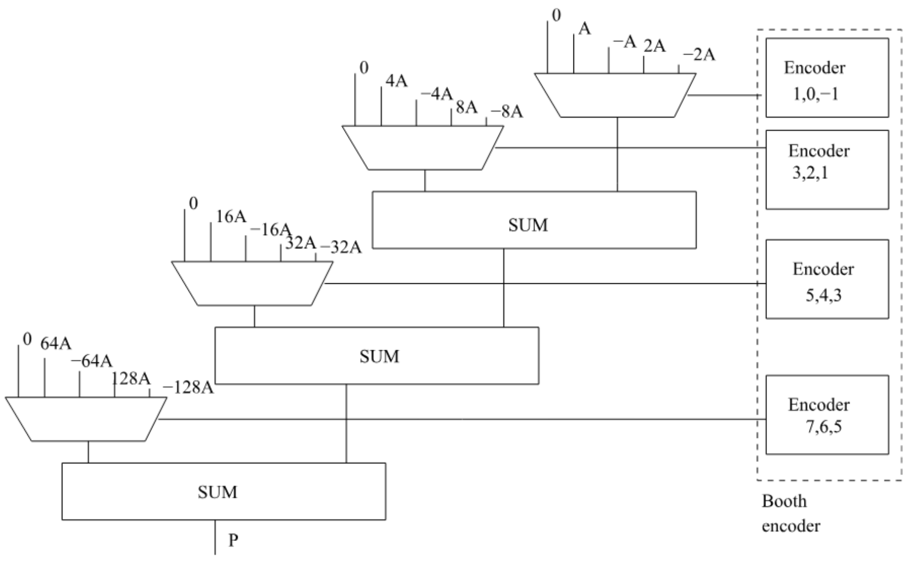
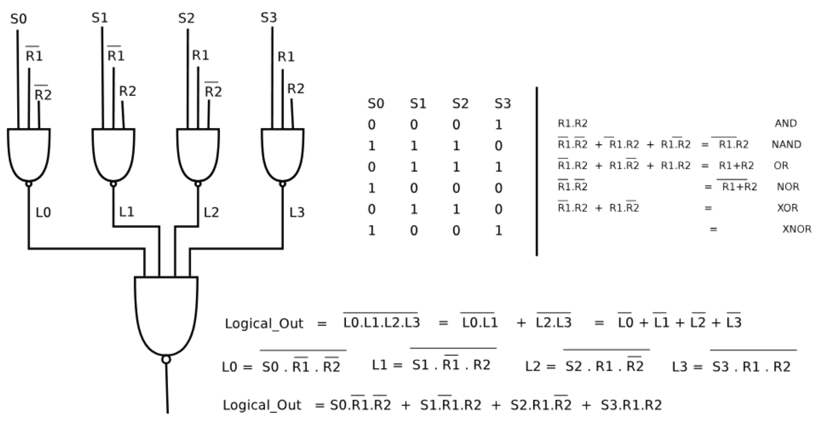


Figure 6: Booth multiplier structure

unsigned comparison flags. To detect equality, the comparator checks if all bits of $A - B$ are zero. Signed comparisons are based on the sign of the subtraction result and the arithmetic overflow condition. The sign bits of A, B, and the result are used to compute the overflow flag, which is then combined with the result's sign to determine whether $A < B$. The remaining signed conditions ($A \leq B$, $A > B$, $A \geq B$) are derived from the previous relation and the zero flag. Unsigned comparisons rely solely on the carry-out from the subtraction. Since $A < B$ in unsigned arithmetic if and only if the subtraction does not generate a carry, the unit can compute unsigned relations directly from the carry bit and the zero flag. Overall, the comparator generates nine outputs: equality, less-than, greater-than, and the corresponding signed and unsigned variants, using only simple combinational logic.

3.2.4 Logic unit

The logic unit is based on the logic unit of the T2, which means that it contains two layers of nand gates.



S0	S1	S2	S3		
0	0	0	1	$R1.R2$	AND
1	1	1	0	$\overline{R1}.\overline{R2} + \overline{R1}.R2 + R1.\overline{R2} = \overline{R1.R2}$	NAND
0	1	1	1	$\overline{R1}.R2 + R1.\overline{R2} + R1.R2 = R1+R2$	OR
1	0	0	0	$\overline{R1}.\overline{R2} = \overline{R1+R2}$	NOR
0	1	1	0	$\overline{R1}.R2 + R1.\overline{R2} =$	XOR
1	0	0	1	$=$	XNOR

$$\text{Logical\_Out} = \overline{L0.L1.L2.L3} = \overline{L0.L1} + \overline{L2.L3} = \overline{L0} + \overline{L1} + \overline{L2} + \overline{L3}$$

$$L0 = \overline{S0 . \overline{R1} . \overline{R2}} \quad L1 = \overline{S1 . \overline{R1} . R2} \quad L2 = \overline{S2 . R1 . \overline{R2}} \quad L3 = \overline{S3 . R1 . R2}$$

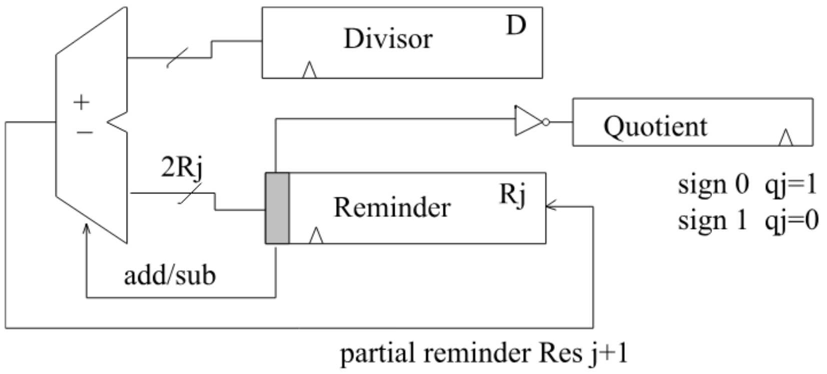
$$\text{Logical\_Out} = S0.\overline{R1}.\overline{R2} + S1.\overline{R1}.R2 + S2.R1.\overline{R2} + S3.R1.R2$$

3.2.5 divider

The divider is implemented using the non-restoring division algorithm. This method iteratively updates the remainder by adding or subtracting the shifted divisor depending on the sign of the current partial remainder. At each step, one quotient bit is generated: a positive remainder selects $+1$, while a negative remainder selects -1. The remainder is then doubled and corrected by subtracting or adding the divisor. After n iterations, the quotient is obtained.This quotient however uses $+1$ and -1 as digits, so it needs to be converted to binary. This approach avoids the need to restore the remainder during intermediate steps, simplifying the hardware and improving performance.

Algorithm 1 Non-Restoring Division

```
1:  $R \leftarrow N$
2:  $D \leftarrow D \ll n$
3:  for $i = n - 1$ down to 0 do
4:      if $R \geq 0$ then
5:          $q(i) \leftarrow +1$
6:          $R \leftarrow 2R - D$
7:      else
8:          $q(i) \leftarrow -1$
9:          $R \leftarrow 2R + D$
10:     end if
11: end for
```



4 Testing

The tests have been conducted using 3 assembly programs:

- **BTB_test**: stores an array in memory, then loads each element and performs several operations (e.g., computing the maximum value, counting the number of negative elements, etc.). This test is designed to verify the correct functioning of the BTB by including a large number of branches (both forward and backward) as well as alternating branch directions.

- **mul_div_test**: a simple program that performs two multiplications in parallel, followed by two divisions where the second division is expected to stall.

- **all_test**: tests the entire instruction set.

Each test verifies the correctness of the operations performed. If all tests pass, a success flag is raised.

5 Physical Synthesis

5.1 Synthesis

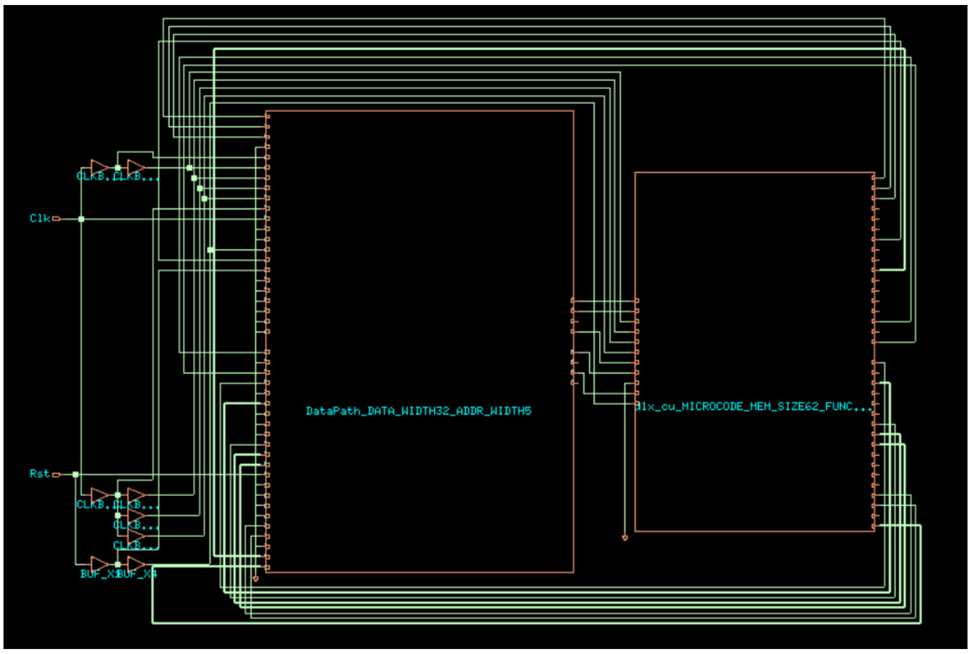

CLKB.
CLKB...
Clk
Rst
CLKB.
CLKB...
CLKB...
CLKB...
BUF_X1
BUF_X4
DataPath_DATA_WIDTH32_ADDR_WIDTH5
dlx_cu_MICROCODE_MEM_SIZE62_FUNC...

Figure 7: Top-level view of the synthesized architecture

The design was synthesized using a script developed for Design Vision. The script analyzes all source files, elaborates a container module that gathers the components to be synthesized, and then generates the timing and power reports along with all files required by the subsequent CAD flow. The container module includes the control unit and the datapath, but excludes both the instruction RAM and the data RAM. The system clock period was set to 3 ns.

5.2 Place and Route

After synthesis, the following steps were performed in the CAD tool to obtain the final layout:

1. **Floorplan definition:** Innovus divides the available area into the region allocated for standard cells and the region dedicated to power distribution.

2. **Insertion of power rings and stripes:** these structures connect VDD and GND to all cells.

3. **Cell placement:** the standard cells are positioned within the defined placement region while avoiding the layers reserved for power delivery.

4. **Signal routing:** the interconnections between cells are created, primarily using the lower metal layers.

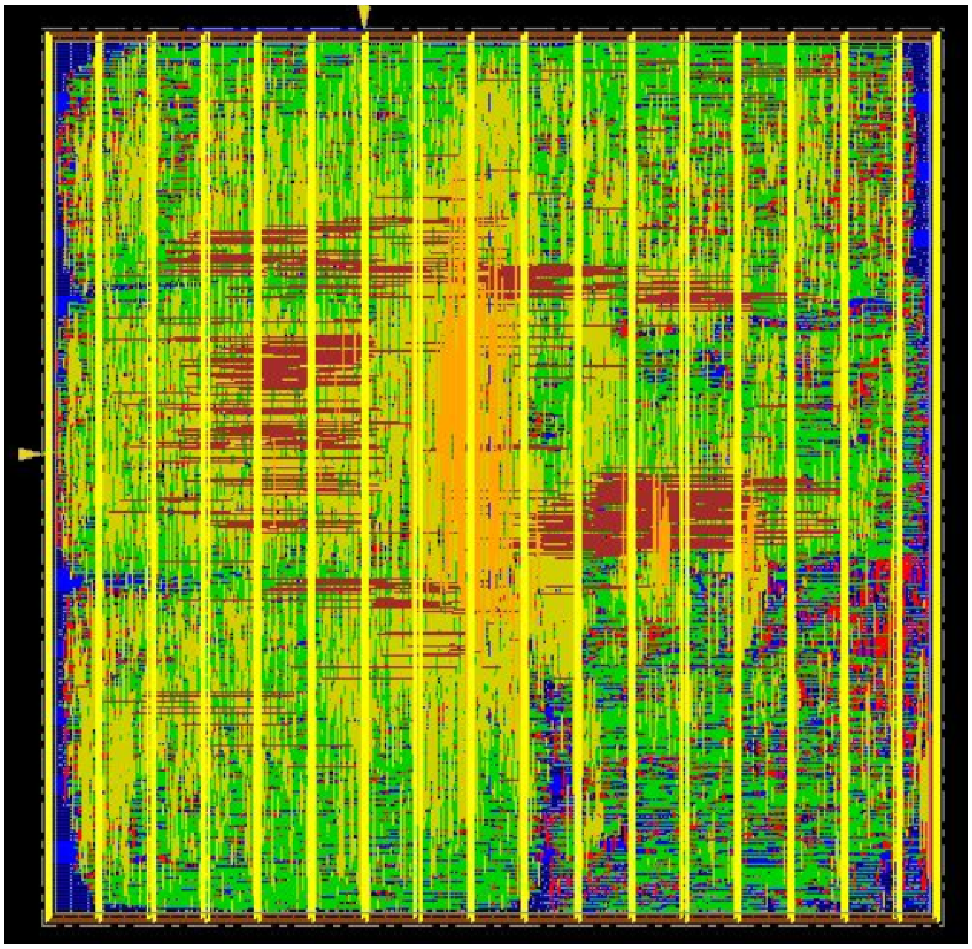


Figure 8: Connections in the final design

5.3 Reports Analisys

- The **Area report** (DLX_NO_RAM.gateCount) shows a relatively large total area of 59 729.5 µm^2. This is mainly due to a small number of modules that dominate the silicon footprint. In particular, the **four largest contributors** are the following:
  - **Branch Target Buffer** — by far the largest block, with an area of 31 551.9 µm^2, representing more than half of the entire design. Its size is due to the large number of entries required for branch prediction.
  - **Register File** — the multi-ported register file requires 9 051.4 µm^2, contributing significantly due to its storage density and multiple read/write ports.
  - **Divider unit** — although included within the ALU hierarchy, it alone accounts for 3 567.9 µm^2 and stands as one of the largest individual computational blocks.
  - **Multiplier unit** — also within the ALU hierarchy, it requires 8 214.1 µm^2 and is the second most area-expensive computational block after the BTB.

Combining these large units (BTB, ALU with multiplier and divider, and the register file) explains the substantial total area of the synthesized design.

- The **Power report** (power.txt) shows that with the chosen clock frequency of 333mhz and the 90nm technology used the total power consumed amounts to 30.93 mW.

  The power is distributed as 19.17 mW (62%) internal, 10.58 mW (34%) switching, and 1.18 mW (4%) leakage. Sequential logic dominates power usage (49%), followed by combinational logic (37%) and the clock tree (14%).

- The **Timing report** (DLX_NO_RAM_postRoute.slk) shows a positive slack for all possible paths, so no violations in these regards. More specifically the lowest slack is 0.593 and this suggests that the clock frequency has room for improvement.