Structural DLX Implementation with Forwarding Logic and Branch Prediction Unit

ROSSI Simone, PoliTO, S228146

PARISI Emanuele, PoliTO, S231543

Contents

Report for Microelectronic Systems Project, Publication date: September 2016.

1. INTRODUCTION

In many places throughout this book we will have occasion to refer to a computer's "machine language". The machine we use is a mythical computer called "MIX". MIX is very much like nearly every computer in existence, except that it is, perhaps, nicer ... MIX is the world's first polyunsaturated computer. Like most machines, it has an identifying number the 1009. This number was found by taking 16 actual computers which are very similar to MIX and on which MIX can be easily simulated, then averaging their number with equal weight:

(360 + 650 + 709 + 7070 + U3 + SS80 + 1107 + 1604 + G20 + B220 + S2000 + 920 + 601 + H800 + PDP-4 + II)/16 = 1009.

The same number may be obtained in a simpler way by taking Roman numerals.

Donald Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*

The DLX is a RISC processor architecture originally designed by John L. Hennessy and David A. Patterson. The DLX is essentially simplified version of the MIPS CPU with a 32-bit load/store architecture (Figure 1 reports the first version of microprocessor based on MIPS Instruction Set Architecture, the `MIPS R2000`).

Like most recent machines, DLX was designed to target:

—A simple load/stor instruction set

—A design for pipelining efficiency, including a fixed instruction set encoding

—An efficiency as compiler target

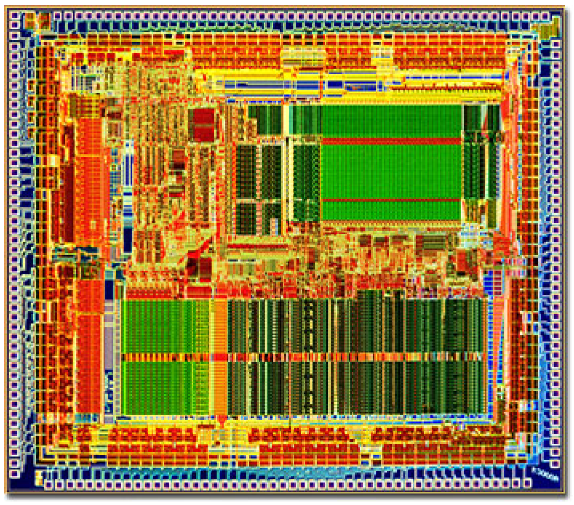

Fig. 1: Chip view of the first microprocessor based on MIPS ISA: The `R2000` (1986)

2. DLX FEATURES

2.1 Registers

DLX has 32 32-bit wide General Purpose Register (GPRs) named `R0`, `R1`, . . ., `R31`; the value of `R0` is hardwired to 0 and it cannot be changed: this allows to execute a variety of useful operation starting from a simple load/store instruction set.

2.2 Data Types

The data types are 8-bit bytes, 16-bit half words, and 32-bit words for integer data. Half words are usually added to the minimal set of recommended data types supported because they are found popular in some programs, such as the operating systems, concerned about size of data structures. The DLX operations work on 32-bit integers: bytes and half words are loaded into registers with either zeros or the sign bit replicated to fill the 32 bits of the registers. Once loaded, they are always operated on with the 32-bit integer operations.

2.3 Addressing Modes

The only data addressing modes are immediate and displacement, both with 16- bit fields. Register pointing is accomplished simply by placing 0 in the 16-bit dis- placement field, and absolute addressing with a 16-bit field is accomplished by using register 0 as the base register. This gives us four effective modes, although only two are supported in the architecture.

2.4 Instruction Format

The instruction format is kept really simple. All operations are encoded on 32 bits with a 6-bit primary opcode and usually they are grouped into three different types: I-type, R-type, J-type (Figure 2.4).

—I-type are normally load and store instructions, operations with immediates or conditional branches.

—R-type are ALU register to register operations, where FUNC defines which is the operations.

—J-type are jump, trap or return from exception.

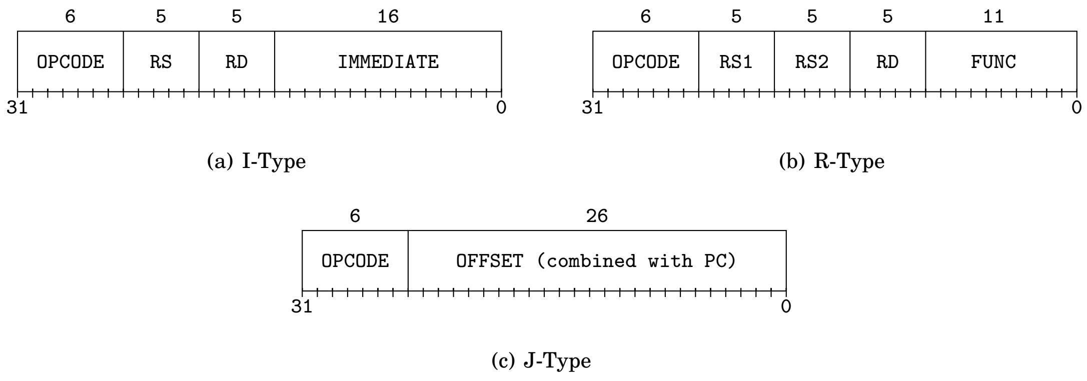

Fig. 2: Instruction format for MIPS 32 ISA

2.5 Operations

In the following table, all the supported operation are listed.

Table I. : List of all the supported operations

add	addi	and	andi	beqz	bnez	j	jal	lw	nop	or	ori	sge
sgei	sle	slei	sll	slli	sne	snei	srl	srli	sub	subi	sw	xor
xori	addu	addui	jalr	jr	lb	lbu	lhi	lhu	sb	seq	seqi	sgeu
sgeui	sgt	sgti	sgtu	sgtui	slt	slti	sltu	sltui	sra	srai	subu	subui

3. BASIC DATAPATH

The Figure 3 presents the general architecture of the DLX. This version of DLX is a 32-bit MIPS-like integer microprocessor based on MIPS32 Instruction Set Architecture.

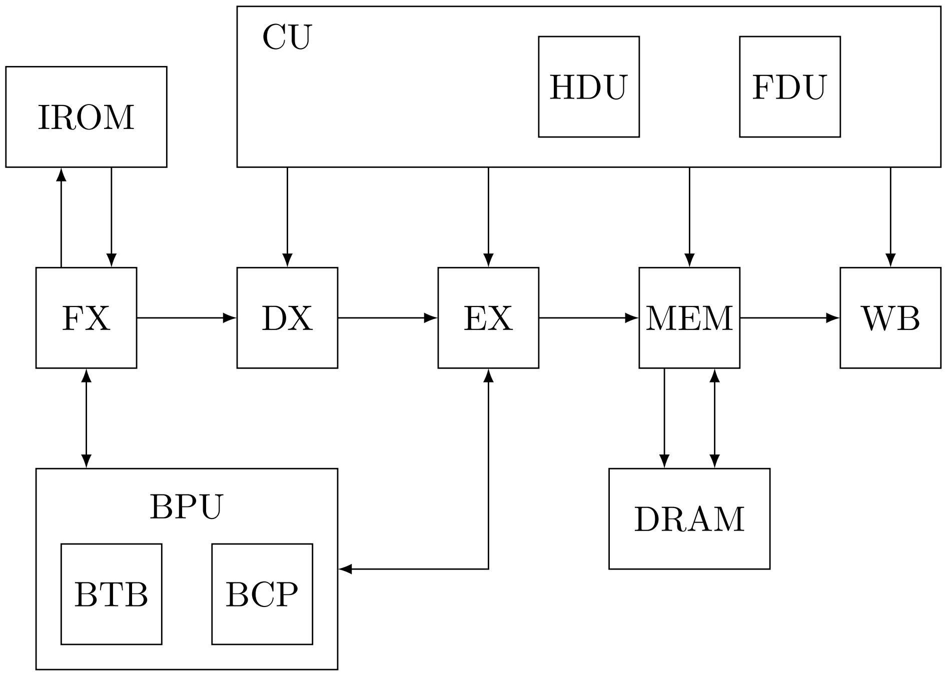

Fig. 3: Basic structure of the DLX

The main datapath is made of five pipelined stages: *Fetch*, *Decode*, *Execute*, *Memory* and *Write back*. During FX the instruction is read from an Instruction Read Only Memory (IROM) and then sent to the DX. During this stage, the instruction is decoded and all data are prepared to be issued. When it is ready, the instruction can move to the real execution: in EX the ALU is configured to process correctly the input data. Finally, during MEM and WB, the result is stored in memory in a Data RAM or inside the on chip register file so that the instruction ends its execution.

The datapath is surrounded by two additional blocks:

—BPU or Branch Prediction Unit, is in charge of reducing control hazards that may arise in a normal flow of execution of programs. It's made of a Branch Target Buffer (BTB) and a Branch Condition Predictor (BCP).

—CU, the Control Unit. It decoded instructions and arrange the datapath according to the type of instruction under execution. One more entity is added to identify possible conditions in which forwarding can be used (thanks to a Forwarding Detector Unit, FDU).

4. ALU ARCHITECTURE

4.1 Adder

The adder implemented on this version of the DLX is a 32 bit Carry Look-ahead Tree-Adder.

As known, the bottleneck of simple adders (like the Ripple Carry Adder) is the generation of the carry-bit. The idea to overcome this issue is to split the addition into two different stages: one for the sum generation and the other one for the carry generation (Figure 4).

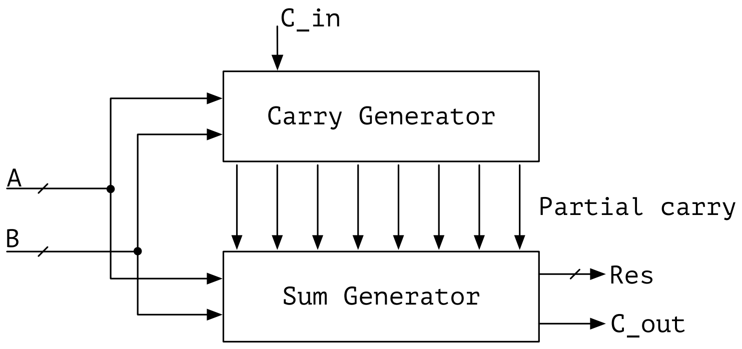

Fig. 4: Basic idea of tree adders

The carry generator is in charge of pre-computing all the carries starting from the two operands without knowing in advance the actual partial-sum, while the sum generator performs the addition (it is likely based on Carry Select Adder).

All possible carry generator are based on the concepts of generation and propagation.

Taking as an example Table II, c_{out} is present at the output in only two cases: either it has been **generated** at bit i^{th} or it has been generated in the previous bit $(i-1)^{th}$ and it has been **propagated** through bit i^{th}.

More formally, generate g_i and propagate p_i are defined in Equation (1) and in Equation (2).

$$p_i = a_i + b_i \tag{1}$$

$$g_i = a_i \cdot b_i \tag{2}$$

Table II.

a_i	b_i	c_{in}	c_{out}	
0	0	0	0	
0	1	0	0	
1	0	0	0	
1	1	0	1	Generation
0	0	1	0	
0	1	1	1	Propagation
1	0	1	1	Propagation
1	1	1	1	Generation

Starting from these definitions, the output carry-bit can be immediately computed as expressed in **??**.

$$c_{out} = g_i + c_{in} \cdot p_i \tag{3}$$

Working bit per bit, the overhead in terms of time and logic complexity might increase with respect to a simple Ripple Carry Adder; this framework, on the other hand, can be easily extended to work with higher radix, considering not just one bit at a time but instead a cluster (a power of 2-bit cluster will be easier to implement).

In this case, the definition of generate and propagate will change to take into account more information (Equation (4) and Equation (5)).

$$G_{i,j} = G_{i,k} + G_{k-1,j} \cdot P_{i,k} \qquad i > k > j \tag{4}$$

$$P_{i,j} = P_{i,k} \cdot P_{k-1,j} \qquad i > k > j \tag{5}$$

The carry bit at position i^{th} now can be written as expressed in Equation (6),

$$c_i = G_{i,k} + G_{k-1,j} \cdot P_{i,k} \qquad i > k > j \tag{6}$$

that can be read as *”carry in position i^{th} is present if it has been generated in the range $[i : k]$ or it as been generated in the range $[k - 1 : j]$ and it has been propagate to the next cluster $[i : k]$”.*

There can be different tree implementation starting from this formal description of carry generation problem. The one used in this version of the DLX is the so called Kogge-Stone adder. With respect to other ones, the Kogge-Stone adder takes more area to implement, but in general it assures the minimum depth and it has a lower fan-out at each stage, which increases performance for typical CMOS process nodes. However, wiring congestion is often a problem for Kogge-Stone adders.Figure 5 illustrates the Kogge-Stone architecture used on the DLX: it works on 32 bits with a carry output every four bits.

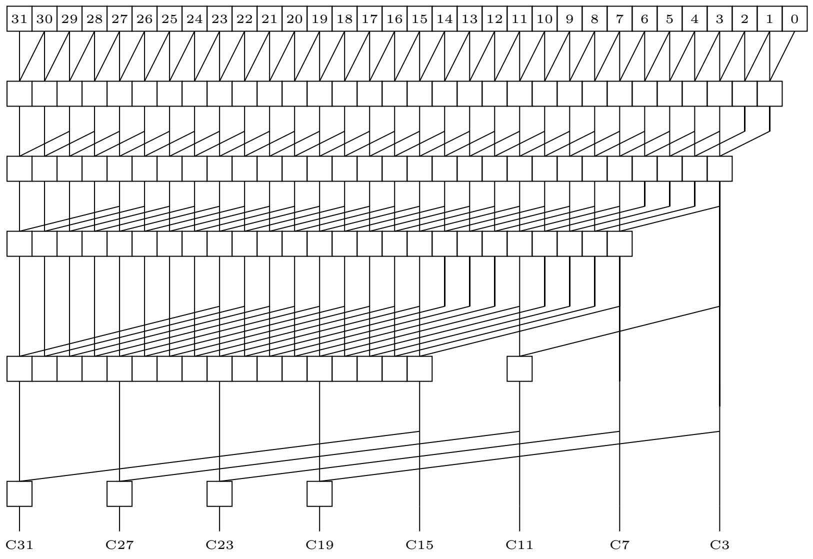

Fig. 5: Tree implementation of the Kogge-Stone architecture with a carry step of 4 bits

The sum generator is made of small group of independent 4 bits adders in an architecture called Carry Select Adder. In a Carry Select Adder (Figure 6), the carry is speculated in both cases ($c_{in} = 0$ and $c_{in} = 1$) and the correct result is then forwarded to the output according to the correct carry bit provided by the above-mentioned carry generator.

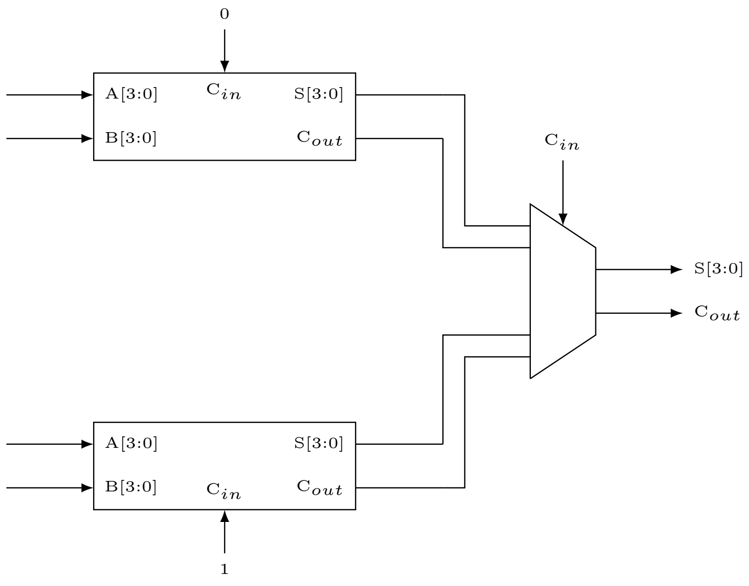

Fig. 6: Basic architecture of the Carry Select Adder

The biggest benefit in term of time performance of this architecture is that the knowledge of the carry is not mandatory for the calculation of the sum: the addition is calculated in both cases and as soon as the carry arrives, the result is ready. On the other hand, this architecture needs two adders, resulting an increasing in the area overhead.

4.2 Shifter

The main idea under the shifter design is to have, given an N-bit shifter, N multiplexer, each of which takes N bits as input and drives a single output bit, such that the amount of shift is given thrught the selection signal of the multiplexer. This is done both for left shift and for right shift, such that a 2 to 1 multiplexer choose the correct result to produce, given a left_not_right signal as selection.

In order to provide both logical and arithmetical shift, a signal is set, which specifies the value of the bit, 0 or 1, to use for filled MSB of the result in case a right shift is requested; this is multiplexer

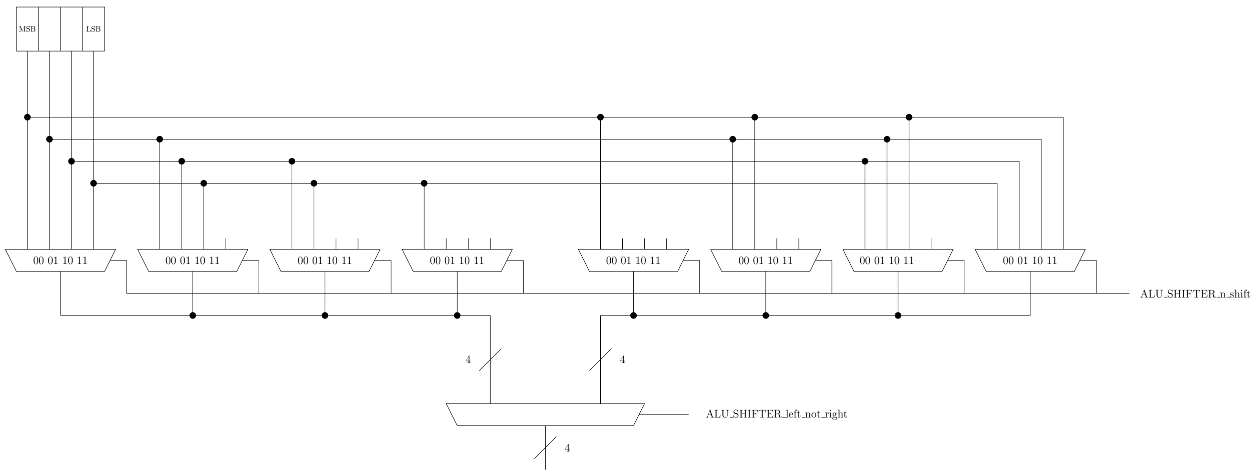

Fig. 7: Basic architecture of a 4-bit shifter

is a simple 2 to 1 multiplexer that choose between the MSB of the operand to shift and 0, given the logic_not_arith control signal, provided as input to the shifter.

4.3 Logical Unit

The logical unit used in this implementation of DLX is a modified version of the logical unit designed for OpenSPARC T2.

The idea behind this architecture is to fuse the logic with the operation selection multiplexer. Therefore, the logic unit is implemented with two NAND gates (nand) level: the first one has four 3-inputs nands, each input is 32-bits wide; the second level has a 4-inputs nand, whose inputs are the outputs of the first level gates (Figure 8). Each first level gate has a select input (extended to 32 bits), while the other two inputs are ta combination of R1 and R2 operands, or their negative value.

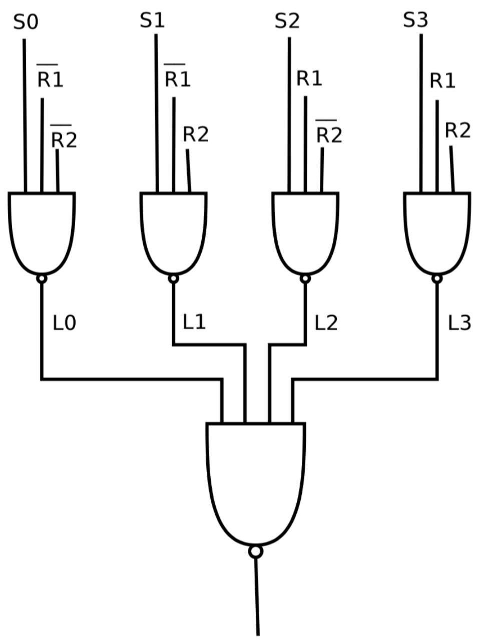

Fig. 8: Basic architecture of our version of the T2 logical unit

S0	S1	S2	S3	FUNC
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

This table reports all the possible valid combination for the operation selection signal. The biggest advantage of this architecture is that is a 2-level logic and that routing and operations are embedded into only one structure.

4.4 Comparator

In our MIPS implementation, the comparator is an independent circuit which does not imply the usage of an adder. It is made by a group of simpler 1 bit comparator arranged in a tree structure.

The single bit comparator, from now on called BIT_COMP, receives as inputs the two operand bits, and two more signals that takes into account the fact that the comparison between two more significant bits already allows to give the result of the comparison. As output two bits are provided, each of them is 1 if the result of the first input bit is greater than the second or if it is lesser than. The two bits are 0 if and only if the two bits in input are equal and the two input signals are both zero.

Below a representation of the BIT_COMP is given.

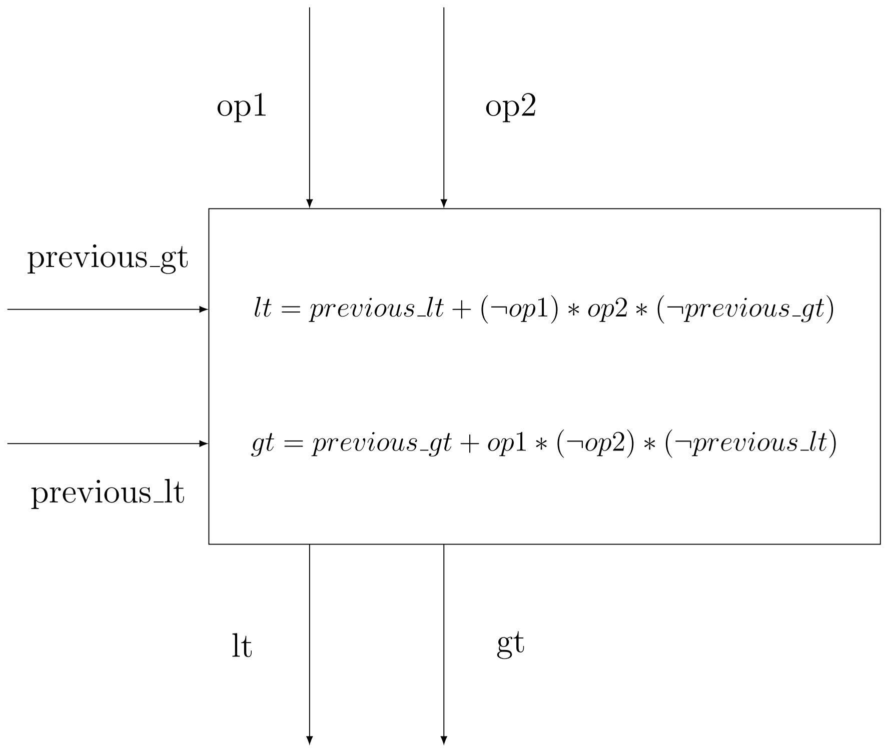
op1
op2
previous_gt
$lt = previous\_lt + (\neg op1) * op2 * (\neg previous\_gt)$
$gt = previous\_gt + op1 * (\neg op2) * (\neg previous\_lt)$
previous_lt
lt
gt

Fig. 9: Single bit comparator

Now we're in a situation very similar to the design of an adder: we have designed a sort of *full adder*, a unit that compute a small part of the final result, given its local input and the results of the previous units that worked on different bits.

From now on we can develop the comparator by following the same principle used when developing an adder: we could connect all BIT_COMP in a row, by emulating a sort of *ripple comparator*, which would correspond to a minimum resources, maximum delay implementation, or we can arrange out BIT_COMPs in a sort of *tree* fashion, that required much more resources to be realized, but allows faster implementation. That's what we've done in our MIPS.

In order to make things easier, we will use a simple 2 bits example.

As you can see from the example, as a first step, each BIT_COMP makes its computation independently from the others. The second step compute the final result, on the basis of the following

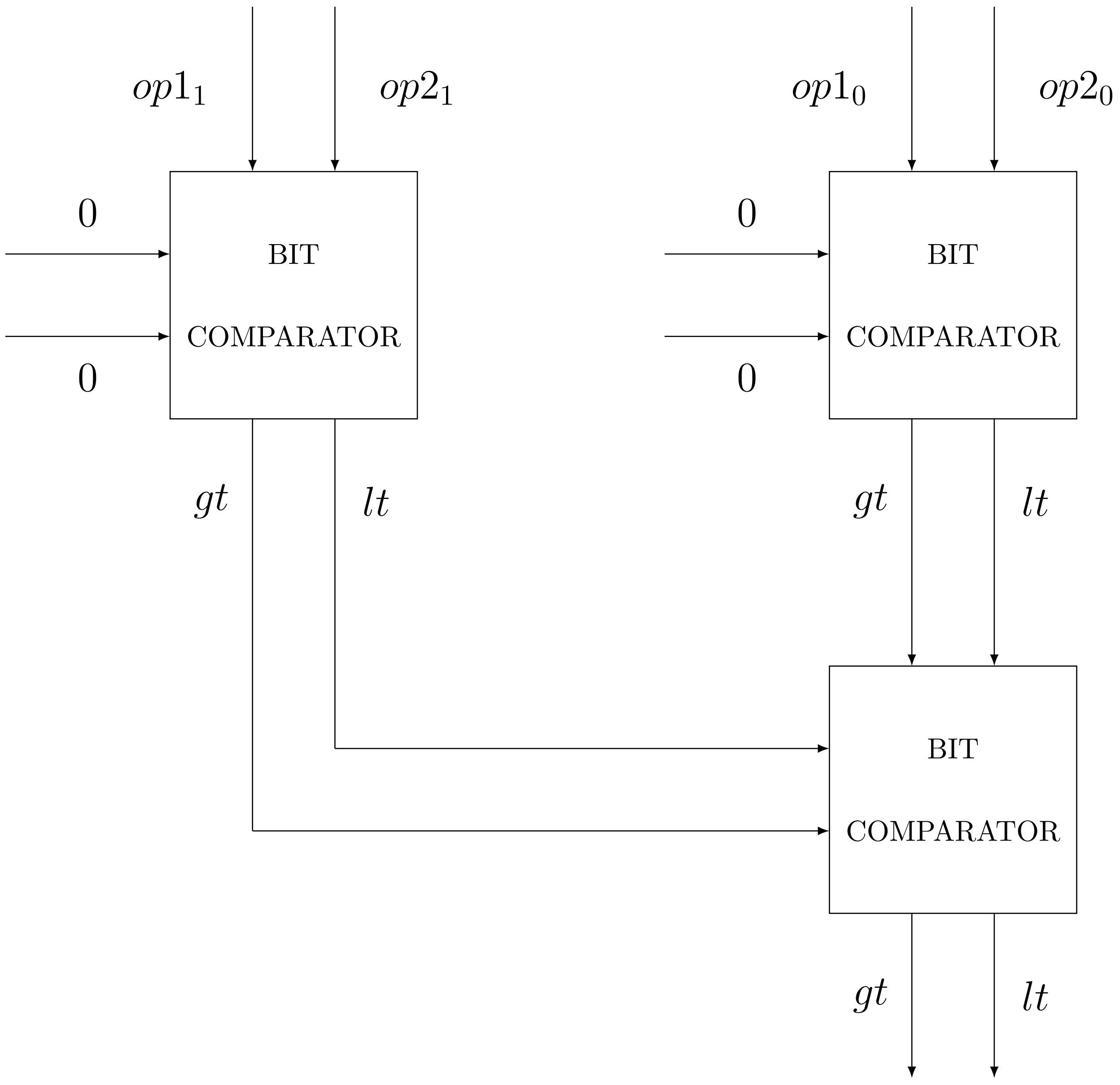

Fig. 10: 2-bit comparator

assumption: the result of the computation is always given by the LESS significant bits, UNLESS a computation performed on a most significant couple of bits gave a result different than equal. This simple example can be generalized on more bits, for example, the 4 bit resulting circuit would be the following:

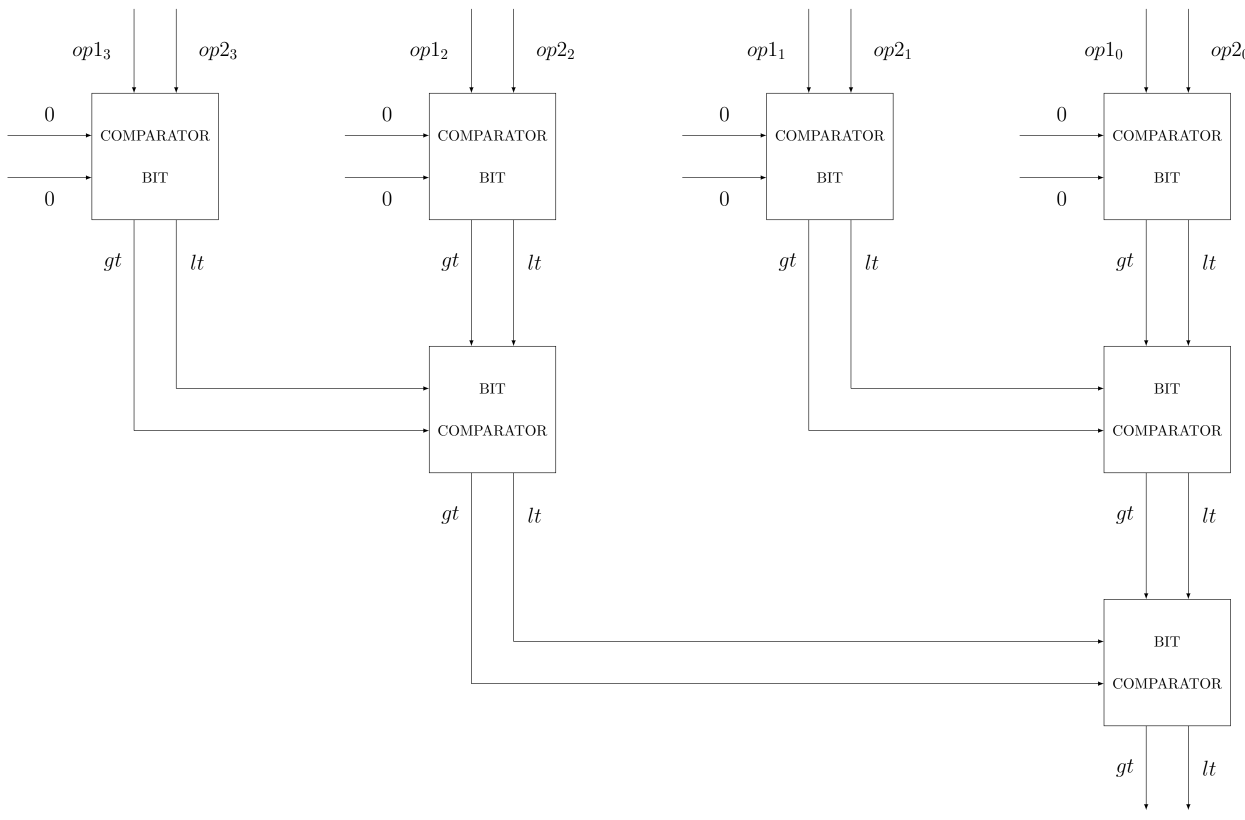

Fig. 11: 4-bit comparator

In our implementation of DLX, a circuit like that ensures a 32 bits comparation in logarithmic time, without implying any subtraction.

The circuit described right now only provides three results: $greater\_than$, $lesser\_than$ and $equal$, but obviouly other results such as $lesser\_equal\_than$, $greater\_equal\_than$ and not_equal are easily computable by the former three.

As a last comment notice than this circuit as it is only provides an unsigned comparation, but it can be converted in a signed/unsigned comparator on the basis of a simple last circuit, which use the following observation: if the result of an unsigned comparation is not $equal$ or not_equal, the result of the signed comparation is simply the opposite of the one computed, if and only if the two most significant bits of the operands are different.

5. DATA FLOW

During the normal flow of execution there are situations, called *hazards*, that prevent the next instruction from executing during its designated clock cycle. For this reason, hazards reduce the performance from the ideal speedup gained by pipelining.
There are three classes of hazards:

(1) *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

(2) *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

(3) *Control hazards* arise from the pipelining of branches and other instructions that change the PC. Hazards in pipelines can make it necessary to stall the pipeline.

5.1 Hazard Detector

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined machine. A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining would change the order of access to an operand.
Consider the pipelined execution of these instructions:

```
ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7
```

The ADD operation will write the final result of addition on during the last stage of pipeline (Write Back); the problem arises because the SUB operation wants to read the content of R1 immediately during the Decode Stage, retrieving a wrong operand (if no further actions are taken).

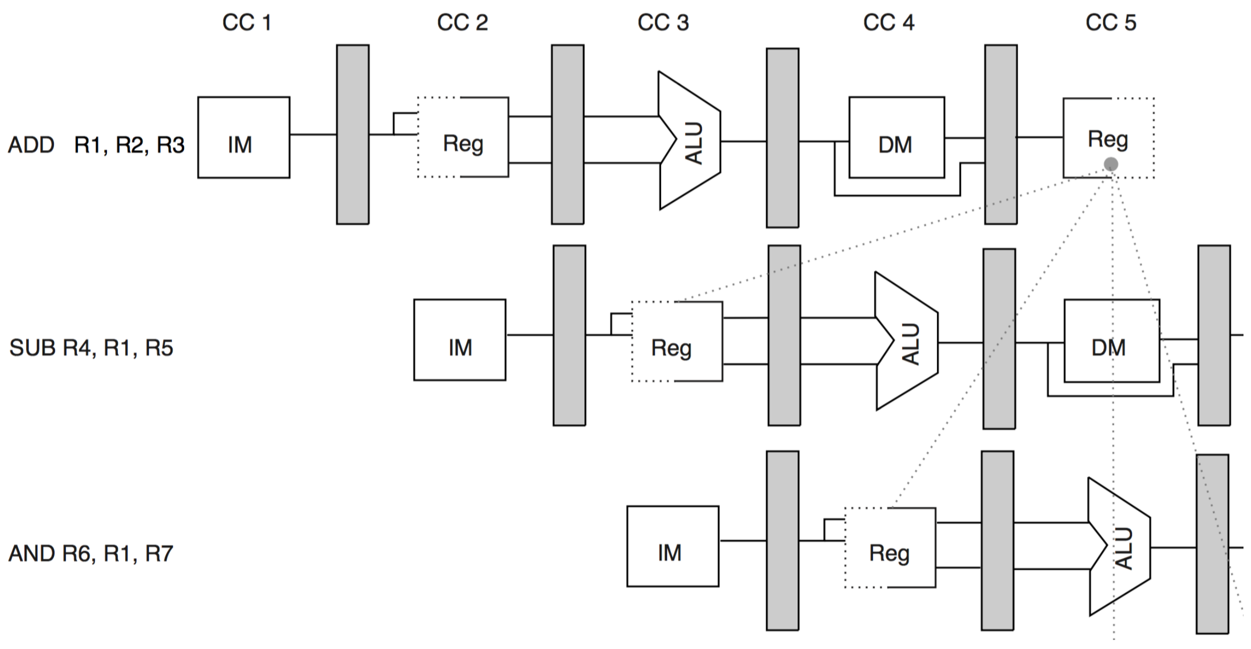

Fig. 12: Graphical representation of pipeline and data hazards

As Figure 12 shows, the SUB and the AND operations read the content of R1 before it's updated by the first ADD.

Two can be the solutions for this issue:

—*Forwarding* some intermediate results to previous stages, without waiting for the result to be correctly written into the register file.

—*Stalling* or inserting a bubble into the pipeline such that the operation already in the pipeline can continue and finish while in meantime no further instructions are issued.

Even if not all possible hazard can be solved by adopting the forwarding logic, the greatest part are: only operation involving memory has to be managed by stalls.

Table III. : Comparison needed to know whenever a forwarding has to be enabled

Source	Opcode source	Dest.	Opcode destination	Dest. result	Comparison
EX/MEM	Register-register	ID/EXE	Register-register, immediate, load, store, branch	Top ALU	$\text{EX/MEM}_{16..20}$ == $\text{ID/EX.IR}_{16..20}$
EX/MEM	Register-register	ID/EXE	Register-register	Bottom ALU	$\text{EX/MEM}_{16..20}$ == $\text{ID/EX.IR}_{11..15}$
MEM/WB	Register-register	ID/EXE	Register-register, immediate, load, store, branch	Top ALU	$\text{MEM/WB.IR}_{16..20}$ == $\text{ID/EX.IR}_{16..20}$
MEM/WB	Register-register	ID/EXE	Register-register	Bottom ALU	$\text{MEM/WB.IR}_{16..20}$ == $\text{ID/EX.IR}_{11..15}$
EX/MEM	Immediate	ID/EXE	Register-register, immediate, load, store, branch	Top ALU	$\text{EX/MEM.IR}_{11..15}$ == $\text{ID/EX.IR}_{16..20}$
EX/MEM	Immediate	ID/EXE	Register-register	Bottom ALU	$\text{EX/MEM.IR}_{11..15}$ == $\text{ID/EX.IR}_{11..15}$
MEM/WB	Immediate	ID/EXE	Register-register, immediate, load, store, branch	Top ALU	$\text{MEM/WB.IR}_{11..15}$ == $\text{ID/EX.IR}_{16..20}$
MEM/WB	Immediate	ID/EXE	Register-register	Bottom ALU	$\text{MEM/WB.IR}_{11..15}$ == $\text{ID/EX.IR}_{11..15}$
MEM/WB	Load	ID/EXE	Register-register, immediate, load, store, branch	Top ALU	$\text{MEM/WB.IR}_{11..15}$ == $\text{ID/EX.IR}_{16..20}$
MEM/WB	Load	ID/EXE	Register-register	Bottom ALU	$\text{MEM/WB.IR}_{11..15}$ == $\text{ID/EX.IR}_{11..15}$

5.2 Forwarding Logic

Implementing the forwarding logic is similar to hazard detector, even if there are more cases to consider. The key observation needed to implement the forwarding logic is that the pipeline registers have to contain both the data to be forwarded as well as the source and destination register fields. All forwarding logically happens from the ALU or data memory output to the ALU input. The forwarding is needed if the destination registers of the IR contained in the EX/MEM and MEM/WB stages are equal to the source registers of the IR contained in the ID/EX and EX/MEM registers.

Figure 5.2 shows the comparisons and possible forwarding operations where the destination of the forwarded result is an ALU input for the instruction currently in EX.

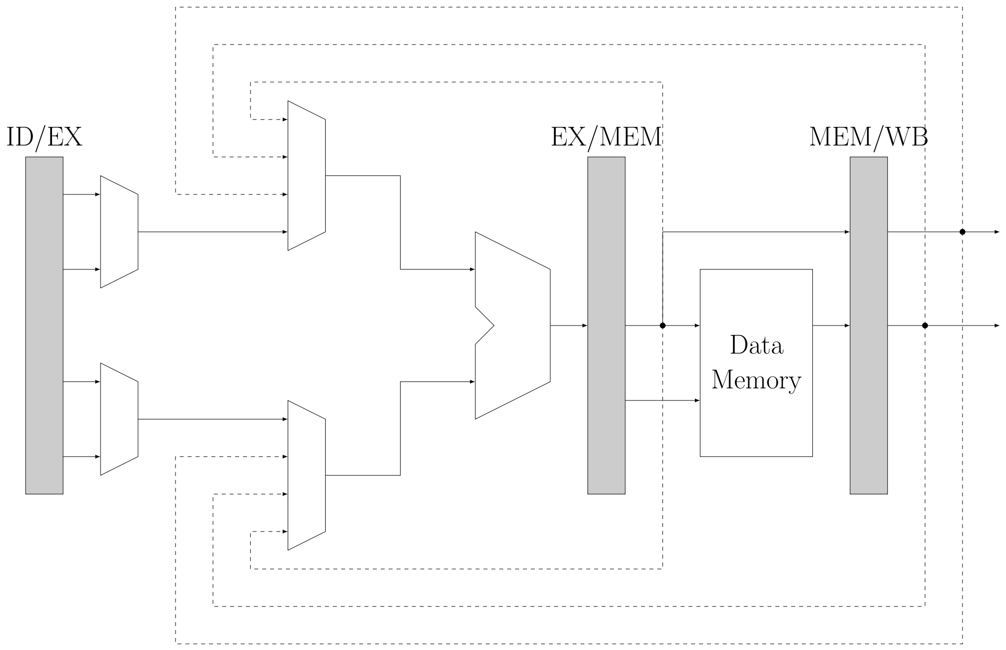

Fig. 13: Pipeline structure of forwarding operation

In addition to the comparators and combinational logic that is mandatory to determine when a forwarding path needs to be enabled, we also need to add further multiplexers at the ALU inputs and add the connections from the pipeline registers that are used to forward the results. Figure 5.2 shows the relevant structure of the pipelined datapath with the additional multiplexers and connections in place.

The second level of multiplexer is controlled by the forwarding unit inside the general control unit.

6. CONTROL FLOW

Control hazards can cause a greater performance loss for our DLX pipeline than data hazards. When a branch is executed, it may or may not change the Program Counter to something other than its current value plus 4.

The simplest method of dealing with branches is to stall the pipeline as soon as we detect the branch until the new Program Counter is determined. However two/three clock cycles wasted for every branch is a significant loss. The number of clock cycles in a branch stall can be reduced by two steps:

(1) Find out whether the branch is taken or not taken earlier in the pipeline or speculate on its condition

(2) Pre-compute the target Program Counter and eventually save it in a memory for the future.

6.1 Dynamic Branch Prediction Unit

In general, the problem of the branch becomes more important for deeply pipelined processors (like DLX) because the cost of incorrect predictions increases (the branches are solved several stages after the ID stage). The main goal of branch prediction techniques is to try to predict as soon as possible the outcome of a branch instruction.

There are many methods to deal with the performance loss due to control hazards:

—*Static Branch Prediction Techniques*: the actions for a branch are fixed during the entire execution

—*Dynamic Branch Prediction Techniques*: the decision causing the branch prediction can dynamically change during the program execution

The basic idea of the dynamic branch predictors is to use the past branch behaviour to predict the future. A special hardware is added to dynamically predict the outcome of a branch: the prediction will depend on the behaviour of the branch at run time and will change if the branch changes its behaviour during execution.

6.2 Branch Predictor

One of the most famous and used predictors is the two-bit prediction scheme based on up/down saturating counters. In a two-bit scheme, a prediction must miss twice before it is changed.
The two-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one half of its maximum value (2^{n-1}), the branch is predicted as taken; otherwise, it is predicted not-taken. As in the two-bit scheme, the counter is incremented on a taken branch and decremented on an not-taken branch.

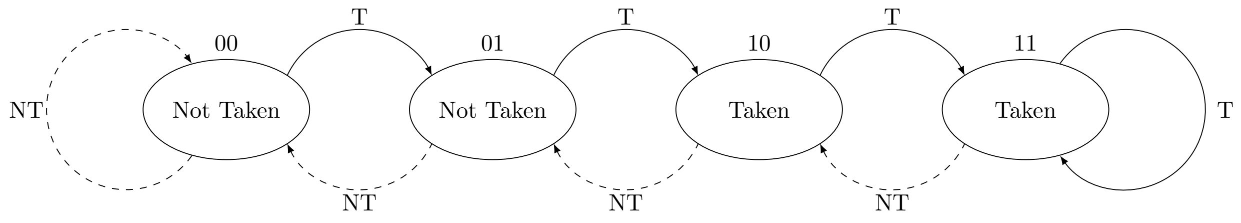

Fig. 14: Structure of the saturating counter

Figure 14 demonstrates how this predictor works: the edges are characterized by the branch result based on the fact that the calculated condition has brought to a taken (T) or not-taken (NT) branch. Inside the bubbles there is the speculated condition for the next execution of the same branch.
It's clear that the behaviour of a branch is changed whenever, in the near past, two predictions were wrong (either taken or not-taken).

7. FINAL DEVICE

After designing every single block hierarchically, the final processor has be built and every each parts connected (Figure 7).

Unfortunately, due to tight deadline, the Branch Prediction Unit has been successfully designed and tested on its own but it's not part of the final device. Nevertheless, the final design already embeds all the logic needed by the BPU and it's matter of verifying that everything works according to the design rules.

All control signals are driven by a classical hardwired *control unit*, which represent the best tradeoff between timing efficiency and designing effort (Figure 7).

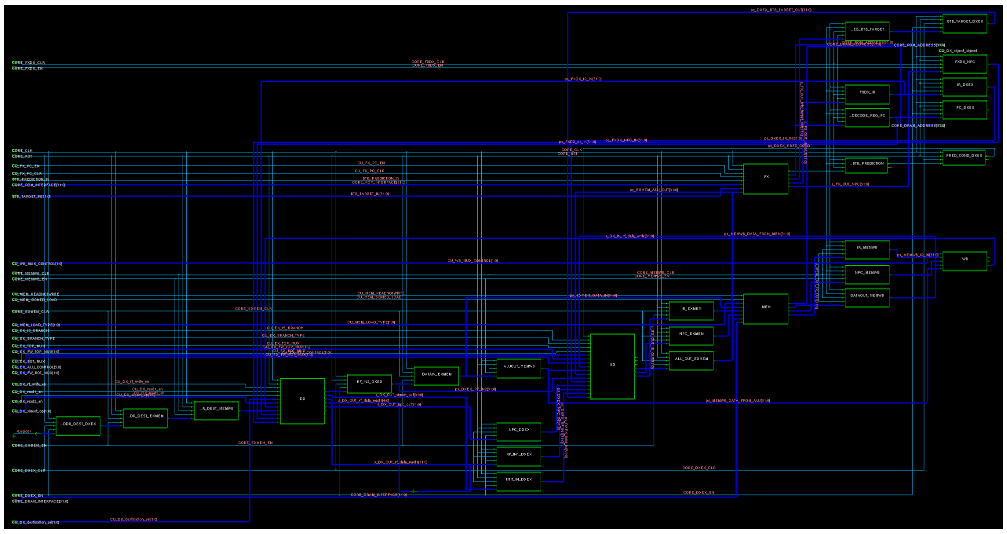

Fig. 15: Full pipelined datapath (image provided after the command `elaborate` by Synopsys DC Compiler)

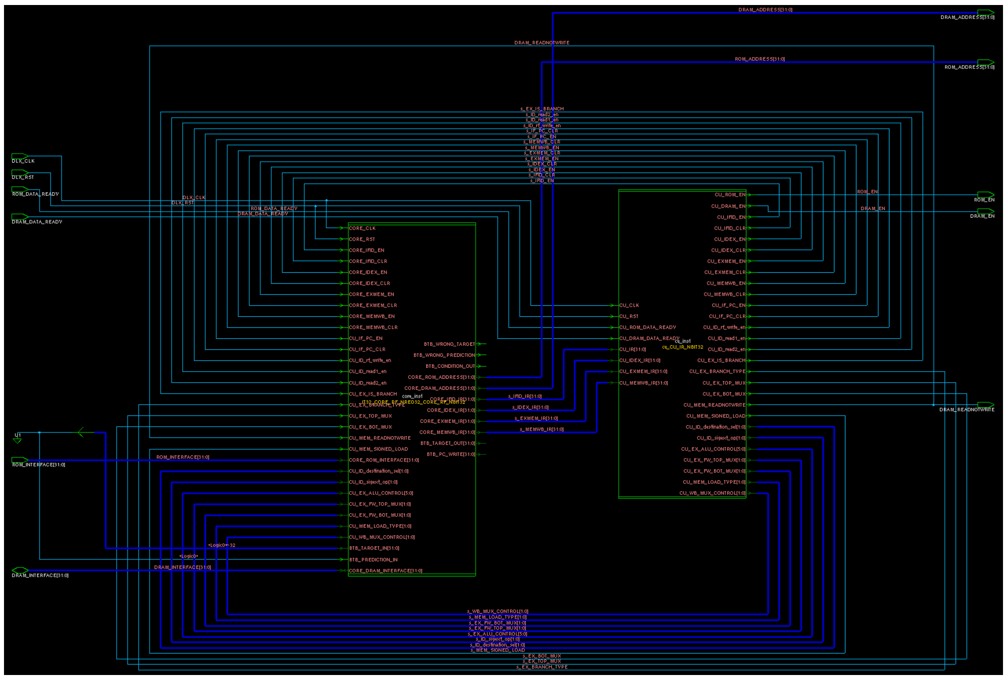

Fig. 16: Datapath surrounded by the Control Unit and memory interfaces (image provided after the command `elaborate` by Synopsys DC Compiler)

8. SYNTHESIS RESULT

The full microprocessor has been compiled twice to target two different cell libraries:

—Cell library from **NanGate**, with a technology process of 65 nm and one Single Threashold Voltage (SVT) in nominal conditions at 25 °C

—Cell library from **STMicroelectronics**, with a technology process of 65 nm and Multiple Threashold Voltage (MVT) in nominal conditions at 25 °C

The architectural synthesis tool used in this project is **DC compiler** (Version Z-2007.03-SP1) by Synopsys, Inc.

Since this framework has been designed highly structured and hierarchical, to enable the compiler to maximise the performances (in terms of power and timing) before the *technology mapping phase* the compilation script runs the command `ungroup -all -flatten`, which ”removes all levels of hierarchy from the current design by exploding the contents of all cells in the current design” (from man page).

8.1 Results with NanGate Library

With this library, the synthesis script has been kept very simple, avoiding to enable internal high performance tools. The results of this compilation are reported on next tables.

Based on this report, we can state that the maximum operating frequency that can be used on this version of DLX is about 630 MHz. The analysis of power report is also interesting: using estimated heuristics, the total Dynamic Power is 12.6368 mW while the Cell Leakage Power reaches a value upto 337.1966 µW

Table IV. : Timing Report with 65 nm NanGate Library

Point	Incremental	Path
clock DLX_CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
core_inst/MEMWB_IR/DFF_29/data_reg/CK (DFFR_X1)	0.00 #	0.00 r
core_inst/MEMWB_IR/DFF_29/data_reg/QN (DFFR_X1)	0.07	0.07 r
...	...	...
core_inst/IDEX_RF_IN2/DFF_20/data_reg/D (DFFRS_X1)	0.01	1.54 f
data arrival time		1.54
clock DLX_CLK (rise edge)	1.58	1.58
clock network delay (ideal)	0.00	1.58
core_inst/IDEX_RF_IN2/DFF_20/data_reg/CK (DFFRS_X1)	0.00	1.58 r
library setup time	-0.04	1.54
data required time		1.54
data required time		1.54
data arrival time		-1.54
slack		0.00

Table V. : Power Report with 65 nm NanGate Library

Cell Internal Power	11.6955 mW	(93%)
Net Switching Power	941.3134 uW	(7%)
Total Dynamic Power	12.6368 mW	(100%)
Cell Leakage Power	337.1966 uW	

8.2 Results with STMicroelectronics Library

With this library, we target high performance both in timing and in power. Regarding power consumption, the developed script works both on leakage power and on dynamic power.

Leakage power is emerging as a key challenge in IC design. Leakage is increasingly exponentially with each technology generation and is becoming the dominant part of total power. Device threshold voltage scaling, shrinking device dimensions, and larger circuit sizes are causing this dramatic increase in leakage. Leakage power minimization is an issue of significant concern in nanometer scale CMOS designs.
One of the possible solution for this issue is the *dual-Vth design*. In the dual-Vth a circuit is partitioned into critical and non-critical paths, and low threshold voltage (LVT) cells are used only for gates in the critical paths. This approach will reduce subthreshold leakage current. It may achieve large leakage power reduction if the circuit contains many gates in non-critical paths. Thus, dual VTH scheme is efficient in realizing high-speed and low-power systems.

Regarding dynamic power, the solution implemented is the so-called *clock gating*. The idea is stop the clock to registers which are not in use during some particular clock cycles (idle conditions).

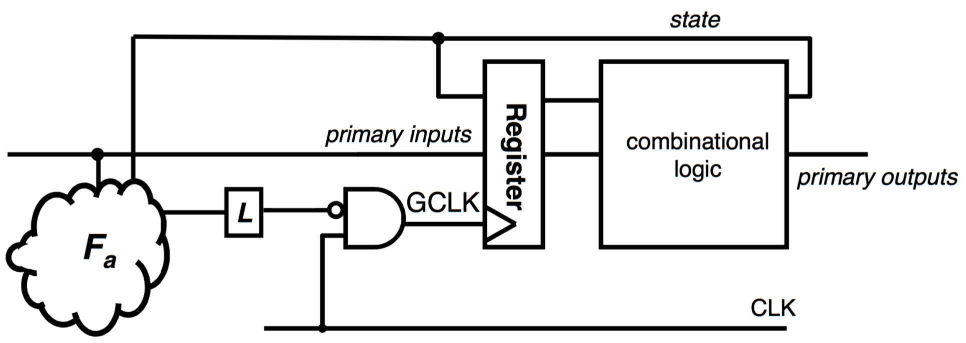

Fig. 17: Clock-gating general architecture

The results of this compilation are reported on next tables.

Table VI. : Timing Report with 65 nm STMicroelectronics Library

Point	Incr	Path
clock DLX_CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
core_inst/IDEX_IR/DFF_28/data_reg/CP (HS65_LL_DFPRQX9)	0.00	0.00 r
core_inst/IDEX_IR/DFF_28/data_reg/Q (HS65_LL_DFPRQX9)	0.12	0.12 f
...	...	...
core_inst/IDEX_IMM_IN/DFF_21/data_reg/D (HS65_LL_DFPRQX35)	0.00	1.33 f
data arrival time		1.33
clock DLX_CLK (rise edge)	1.43	1.43
clock network delay (ideal)	0.00	1.43
clock uncertainty	-0.01	1.42
core_inst/IDEX_IMM_IN/DFF_21/data_reg/CP (HS65_LL_DFPRQX35)	0.00	1.42 r
library setup time	-0.09	1.33
data required time		1.33
data required time		1.33
data arrival time		-1.33
slack		-0.00

Table VII. : Power Report with 65 nm STMicroelectronics Library

Cell Internal Power	7.2549 mW	(79%)
Net Switching Power	1.8754 mW	(21%)
Total Dynamic Power	9.1303 mW	(100%)
Cell Leakage Power	32.0460 uW	

With this library and with a more complicated script, we can achieve better performance, both in timing and in power. The maximum operating frequency is now about 700 MHz and, with respect to the previous value, this one is more realistic: the computation works with a more complex models of the clock (with uncertainty and skew), wires and in/out ports.

Table VIII. : Clock Gating Report with 65 nm STMicroelectronics Library

Number of Clock gating elements	38	
Number of Gated registers	1012	(67.20%)
Number of Ungated registers	494	(32.80%)
Total number of registers	1506	

Table IX. : Threshold Voltage Group Report with 65 nm STMicroelectronics Library

Threshold Voltage Group	Number of Cells	Percentage
LVT	4546	54.47%
HVT	3800	45.53%

Furthermore, both the techniques used to reduce power work as expected. On one hand, about 45% of total cells have been swapped to an high threshold voltage, this means that without modifying the timing constrains, we achieve more than 90% of leakage reduction. On the other hand, clock gating reduces the total dynamic power of about 25%, applying automatically this technique to the 67% of total registers in the design.

9. CONCLUSIONS

To sum up, we designed a full structural and hierarchical DLX, implementing high efficiency architectures in the Arithmetic Logic Unit, which have brought, finally, to an high-performance low-power microprocessor with a realistic operating frequency of 700 MHz. The system has been successfully tested with simple programs to confirm that everything works according to the design rules.
Unfortunately, we had not time to connect the Branch Predictor Unit and verify that the CPI (Clock Per Instruction) reduces.
Further modifications of the project for the future include:

—Testing and verifying the correctness behaviour of the BPU,

—Compilation and synthesis targeting a more recent technology process (for example 15 nm),

—Embed in the final design also the memories, using a memory compiler (for example DesignWare Memory Compilers by Synopsys).

Our DLX after physical design