DLX

October 31, 2017

Contents

1 Introduction **2**

2 Implmentation Overview **2**

3 Design Architecture **3**

3.1 Fetch Block 3
3.2 Branch Predictor 5
3.3 Decode Block 6
3.4 Execute Block 6
3.5 ALU and Multiplier 8
3.5.1 Shared Adder 9
3.5.2 Multiplier Logic 10
3.5.3 Logic Unit 11
3.5.4 Comparator 12
3.5.5 Shifter 13
3.6 Memory Block 14
3.7 Forwarding Logic 15
3.8 Control Unit 15
3.9 Stall Logic 17
3.10 Instruction and Data Memories 18

4 Simulation and Test **18**

5 Synthesis **19**

5.1 Power Optimization 20

6 Layout **22**

7 Conclusions **24**

1 Introduction

The goal of this project is to implement an RTL completely functional DLX processor in VHDL, simulate it and then proceed with the design flow with synthesis and layout phases.
DLX processor architecture was designed by John L. Hennessy and David A. Patterson[3] and very well described and documented all over the internet.
This project is developed for the course Microelectronic Systems taught by Prof. Mariagrazia Graziano in Politecnico di Torino[2].

Outline The remainder of this article is organized as follows:
I will first give a brief overview of the DLX architecture and my personal implementation in Section 2. In Section 3 will go into details regarding particular aspects of the datapath and control unit. Section 5 and 6 are about the environment and results of design and layout phases. Finally, Section 7 gives the conclusions and possible future work.

2 Implmentation Overview

DLX is a classic 5-stage pipelined processor, with fixed length instructions and in-order execution. Most information regarding the ISA, structure and description of operations can be found in [1].

Instructions In this version, all I-TYPE and R-TYPE operations have been implemented with the exception of: LHI, RFE, TRAP, ITLB, LB, LH, LBU, LHU, LF, LD, SB, SH, SF, SD and all MOV operations.
The only F-TYPE instructions implemented are MULT and MULTU, but they operate on integer register instead of floating point ones.
During normal conditions, all operations take exactly 5 cycles to execute, with the exception of multiplications, which require a fixed amount of 14 cycles (10 cycles are spent in the execute stage) and they're not pipelineable.

Jump and branch Jump and Branch instructions target address is aggressively evaluated in the decode stage in order to have less cc penalty. In addition to this, a 2-bit predictor scheme has been implemented. This will be discussed more in the next section.

Control logic The execution is regulated by an hardwired CU, in charge of producing control words for each stage of the pipeline, as well as stall signals to stop the execution when needed. All possible hazards have been solved through a forwarding logic, when this is not enough, a stall signal is sent from the control unit to the registers that need to be stalled.

Memories Two separated memories for instruction and data have been included in the design, but they are actually not part of it because they are always external peripherals.

- Instruction memory is asynchronous, with 1 read port. Write is not possible.
- Data memory is synchronous, with 1 read port and 1 write port. Concurrent write and read is not possible.

Register file Register file has 2 read port and one write port. Concurrent write and read on the same register produces a redirection of the input data to the output.

3 Design Architecture

As previously mentioned, the processor is described in VHDL. Most blocks are written in a structural approach up to very basic logic elements such as multiplexers, half adders, etc ...
An overview of the processor architecture is shown in FIGUREXXX at the end of this document.
In the next sections we are going to see in details each single component.

3.1 Fetch Block

Fetch block is in charge of correctly fetch the instruction to be executed from the memory. This block is tightly coupled with branch predictor. As can be

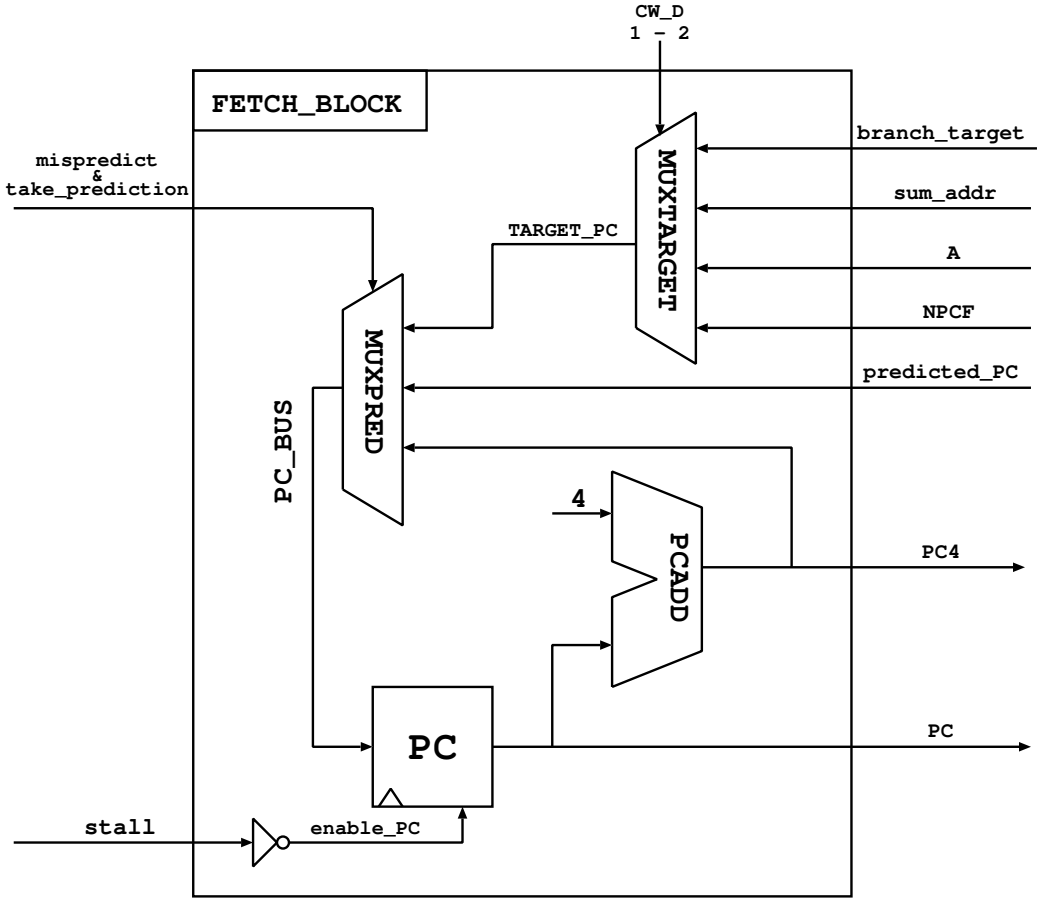

Figure 1: Fetch Block Schematic

seen in the picture, the actual decision of the next PC to drive comes from the BTB in case of normal operations.
MUXTARGET computes the correct target pc, which is sent to the branch predictor to evaluate the misprediction signal that controls MUXPRED. When misprediction signal is triggered, the current operation in the PC is fetched but not sent to decode stage, and it is replaced by the correct one on the successive clock cycle.

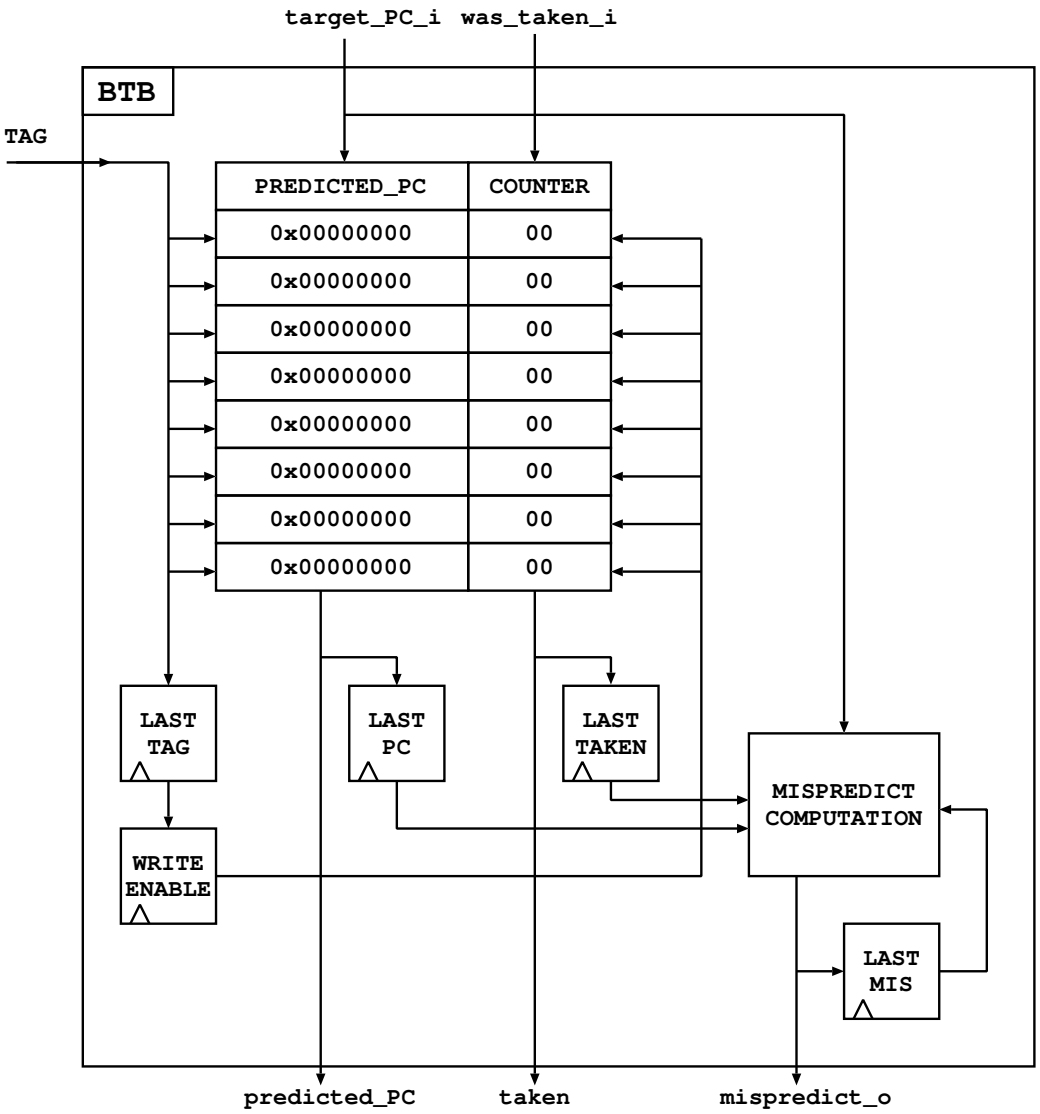

Figure 2: BTB

3.2 Branch Predictor

As previously stated, a branch predictor scheme has been applied, in particular it is a 2bit saturating counter BHT, also widely known in literature as (0,2). In addition, we have also implemented a BTB in charge of storing the PC of the next instruction in case the branch is taken.

The design applied is very generic, so the number of entries is easily configurable.
In normal situations the lowest part of the PC, called TAG, is used to index the BTH, if the instruction is recognized as a taken branch, then the PC found inside the BTB is sent to the fetch block. On the successive clock cycle, when the decode block actually computed the correct behavior of the branch, the value of the BTH is updated and, if the prediction was not correct, mispredict signal is triggered. Also, due to the fact that mispredict signal actually cause the current operation in the fetch stage to be re-evaluated, the branch predictor automatically disables itself on the following clock cycle to avoid possible deadlock conditions.
Some information regarding the operations performed on the last clock cycle need to be stored, to do so, 4 have been added, they store: prediction, prediction target, TAG and mispredict signals.

3.3 Decode Block

Decode block is in charge of 4 main functions:

- Split from IR register number values rA, rB and rC used in the next stage and for forwarding.
- Extend the immediate to 16/26 bits, either signed or unsigned, according to the current decoded instruction.
- Compute the addition between NPCF and the offset given by the extended immediate value
- Evaluate branch output through the zero comparator, the negation of this signal is also sent to other components in the system.

In case of Jump operations with link (JAL and JALR) the immediate value is replaced by the return address by MUXLINK.

3.4 Execute Block

Execute block is third stage of the pipeline. Most of the computation done here is due to the ALU which will be explored in detail in the next section. For the moment let's also forget about forwarding multiplexers, as they are going to be described later.

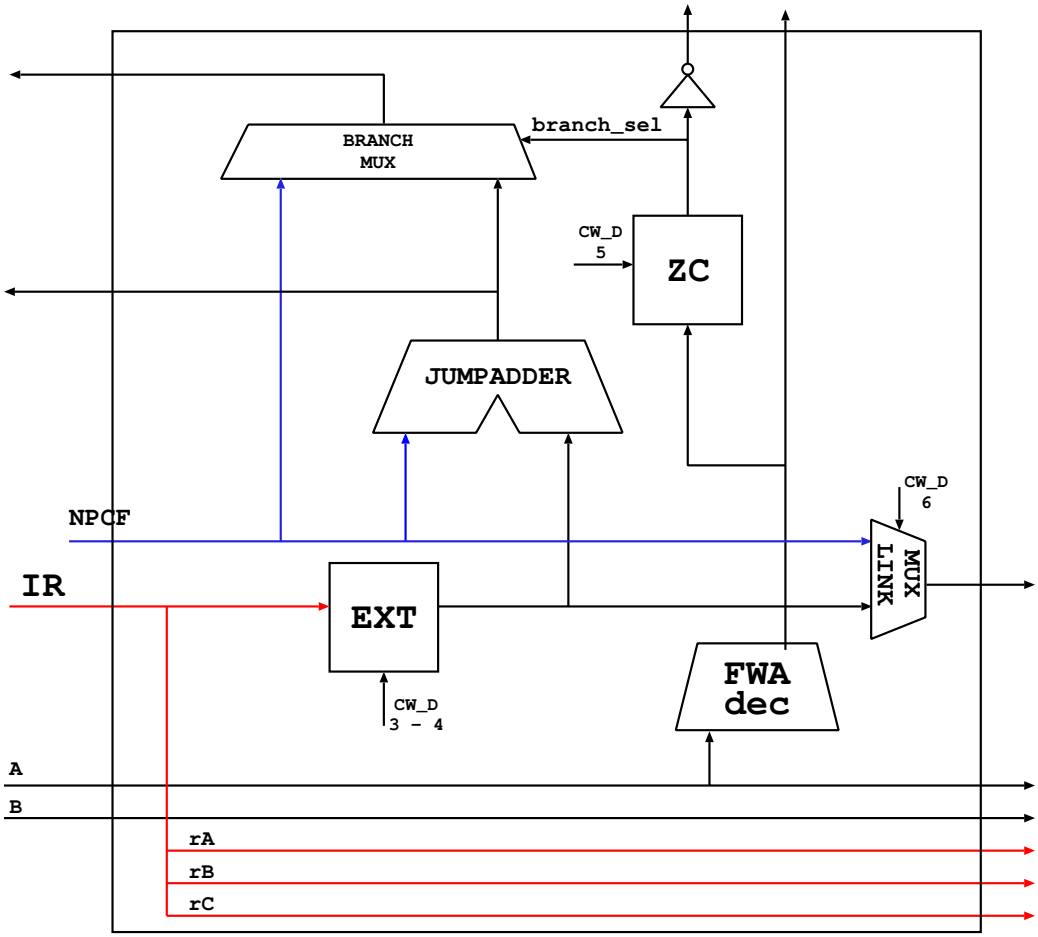

Figure 3: Decode Block Schematic

ALU is fed with two operands: one is always A, while the other could come wither from B or IMM, depending on the type of operation. A multiplexer is then added to achieve this functionality. The multiplexed value is also sent directly to output, it is going to feed the Data memory input in the next stage.
Another multiplexer (DESTMUX) is also used to select in which destination register the current operation is going to write. Again, there is difference between I-TYPE and R-TYPE operations, and also a special input is due to JAL and JALR instructions that have R31 as a forced destination register.

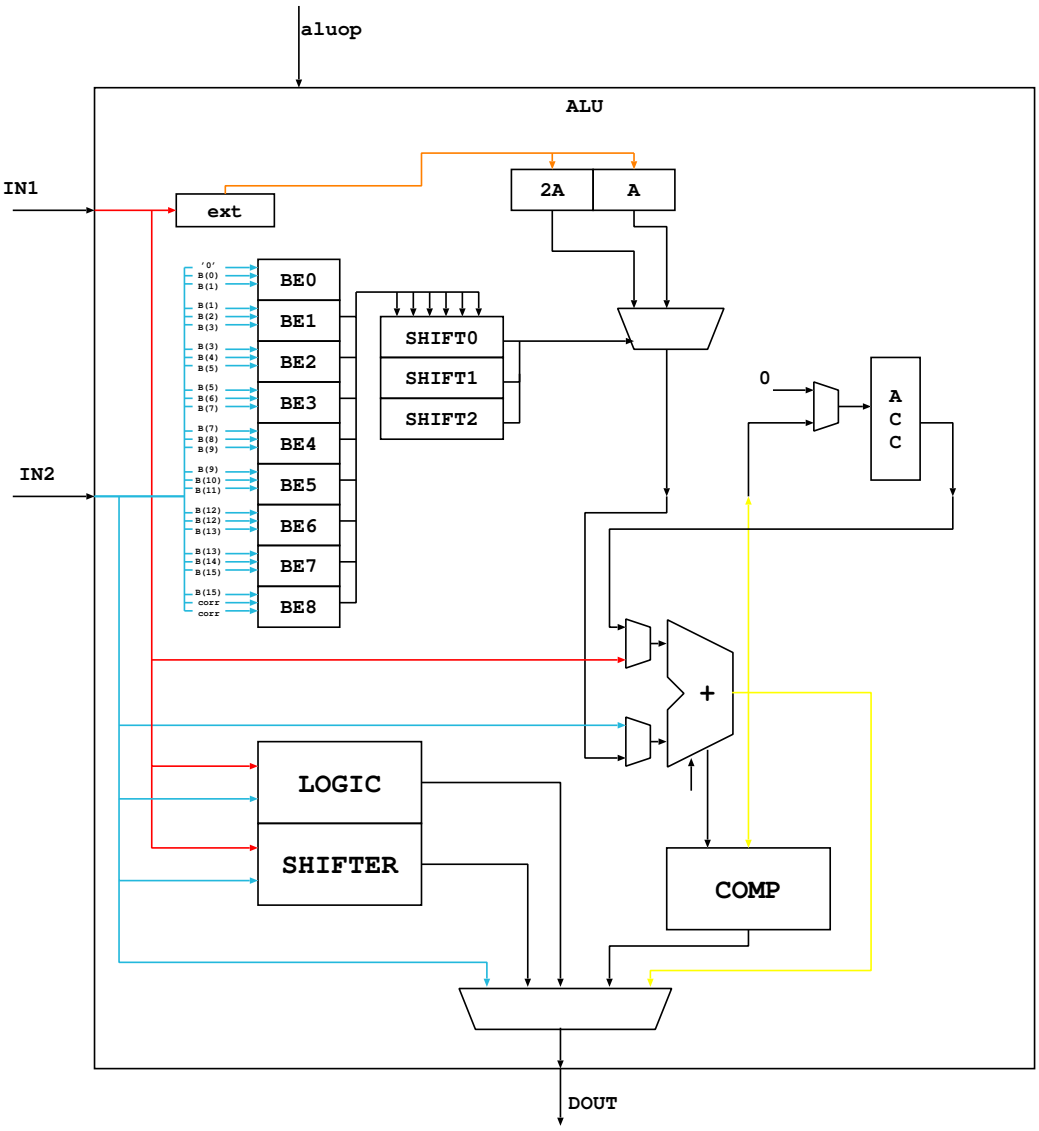

Figure 4: ALU and Multiplier Block Schematic

3.5 ALU and Multiplier

The ALU is responsible for all data manipulation and it's the heart of our processor.
It takes as input two 32 bit operands and a special sequence of control bits generated in the decode stage. The ALU is not directly fed with the ALU

opcode for performance reasons (This was a bottleneck in an early version of the ALU, and there is no need to generate the control bit in the execute stage). The output consist in the 32 bit result and a signal that is at logic '1' during multi cycle operations.
Carry out has been suppressed because in this architecture is never used (unless for comparison operations).
The ALU is composed of the following units:

- A single shared adder
- Multiplication logic
- Logic Unit
- Comparator
- Shifter

The results coming from these units is then multiplexed to the final output. We are going to see in details each of them in the next sections.

3.5.1 Shared Adder

The adder/subtractor is a 32-bit Pentium 4 adder.
It is composed of two main blocks:

Sparse tree The Sparse tree is in charge of computing all carries in a parallel fashion. In our case it is a radix-2 carry-merge tree and generates every fourth carry.

Sum generator The sum generator block is then based on carry select adders, which compute both results in case the carry is 0 and in case the carry is 1. These two addition are independent from the actual carry, so they can be performed in parallel. When the carry reaches the sum generator it controls a multiplexer used to select one of the two possible outputs. As a consequence, the delay due to carry propagation is strongly minimized.

In order to perform subtractions, a little modification has been applied. From basic computer arithmetic we know that subtracting is the same as doing an addition with the second operand being 2's complemented.

Input	Encoding	Action
000	000	+0
001	100	+A
010	100	+A
011	101	+2A
100	111	-2A
101	110	-A
110	110	-A
111	000	0

Table 1: Modified Booth Encoding

2's complement involves doing a not of the entire number and adding 1. Flipping of the bits is performed with a xor circuit placed at the input of the second operand, while the 1 addition is done trough the carry input. (carry input would never been used in this specific architecture).

3.5.2 Multiplier Logic

As briefly introduced, Multiplication is performed by the shared P4 adder in an iterative way. The entire procedure is based on the Booth's multiplier. Instead of producing all the sums in parallel, we compute one each clock cycle. To avoid using 64 bits registers to store a 32x32 multiplication, only the lowest 16-bits of the two operands are considered.
The flow is as follows: At the first clock cycle all the Booth's encoding is done and sent in a PISO register, also value of A is saved into a register able to left shift it's content of 2 every clock cycle (multiplying by 4). Accumulator register is cleaned and set to zero.
Then at each clock cycle the adder takes as input the correctly shifted input value and the result of the previous addition stored in the accumulator register. When a subtraction needs to be performed, the carry input of the adder is set to '1'. To have both signed and unsigned multiplication, a correction stage needs to be performed at the end, as can be seen in the picture, the last encoder takes this into account.
Please also note that a little modification to the booth encoding pattern has been made in order to have a direct mapping between bits and function. Have a look at Table 1.

Bit explanation (MSB to LSB):

- First bit tells us whether or not we actually need to use the adder. This bit is directly sent to the enable of the accumulator, if it is '0', there is no need to sample the new value received from the adder.
- Second bit tells to the adder if it need to compute an addition or a subtraction. This bit is sent both to the sign and the carry in of the adder. '0' stands for addition, while '1' is for subtraction
- Finally the last bit decides the input of the adder. '0' is for the clean value straight of A register, while '1' is used when we need the value left-shifted by 1 position. This bit controls INPUTMUX.

This design is able to reduce critical paths flowing to a standard parallel multiplier and save area as well at the cost of a multi cycle operation.
We put a lot of effort into optimization for this part of the design. The first image shows the initial design, while the latter the final version. Please also consider that we decided to move from 9 to 10 cycles to reduce the critical path. A possible future improvement could be to pipeline the Booth Encoder block too to save area.
We have thought a lot of possible ideas to improve this design from a power-reduction point of view. Unfortunately, most of the possible improvements require latches, and these kind of components are not simple to handle, both for simulation and synthesis aspect.
This is due to the asynchronous behavior of latches and the possible race conditions that might happen in a circuit with latches and flip-flops. We tried something, unfortunately all our efforts achieved nothing and we discarded the idea.

3.5.3 Logic Unit

The DLX architecture supports only 3 type of logical operations: AND, OR, XOR. Due to this simple behavior, we decided to avoid using particular logic unit schemes: the three operations are executed in parallel and then a multiplexer selects the correct one.
As can be see in the picture, the schematic is pretty straightforward.

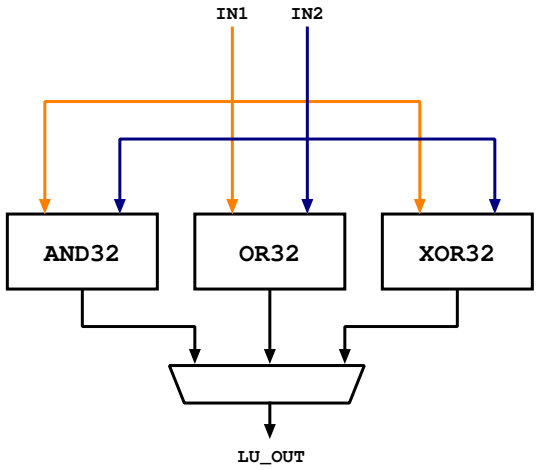

Figure 5: Logic Unit Schematic

3.5.4 Comparator

The ISA requires to implement several comparison operation: $(<, >, \leq, \geq, =, \neq)$. Dealing with both signed and unsigned numbers is not easy in a comparator, because the two operations use different inputs.
For a comparison the required signals are:

- difference between the two numbers, provided by the shared ALU (SUM)
- carry out of the operation (C)
- overflow bit (V)
- sign

Overflow bit is computed outside of the comparator as (IN2(31) xnor SUM(31)) and (IN1 xor IN2). Which basically mean that an overflow is triggered when subtrahend and result have the same sign when minuend and subtrahend have different sign.
Zero bit (Z) is computed through nor reduction of SUM, while the sign of SUM (N) is simply the most significant bit.
There might be some redundancies in the design, we left possible optimizations to the synthesizer.

OP	Description	Flags tested
SEQ	Equal	Z
SNE	Not equal	not Z
SGT	Signed greater	N xnor V
SGE	Signed greater or equal	(N xnor V) or Z
SLT	Signed less	(N xor V) and (not Z)
SLE	Signed less or equal	N xor V
SGTU	Unsigned greater	C and (not Z)
SGEU	Unsigned greater or equal	C
SLTU	Unsigned less	not C
SLEU	Unsigned less or equal	(not C) or Z

Table 2: Comparison Signals

3.5.5 Shifter

The DLX-basic shift operations are: SLL, SRL, SRLI, SRA and SRAI, actually our hardware would be able to implement also the missing ones. Our implementation is based on the T2 coarse-grain shifter and is organized on three different blocks:

- Level 1: this block produces three masks according to the selection signal (00, 01 or 10). shifted by 0 (mask0), 8 (mask8) and 16 bits (mask16) each. Moreover, operand IN1 is extended to 39 bits.

  00 : the operand is shifted left replacing vacant positions with 0s and extended appending 7 0s to the right.

  01 : the operand is shifted right replacing vacant positions with 0s and extended appending 7 0s to the left.

  10 : the operand is shifted right with sign extension and extended replicating the sign bit.

- Level 2: A multiplexer which has as selection signal bits 3 and 4 of operand IN2 does the selection of one of the masks produced in the first block.

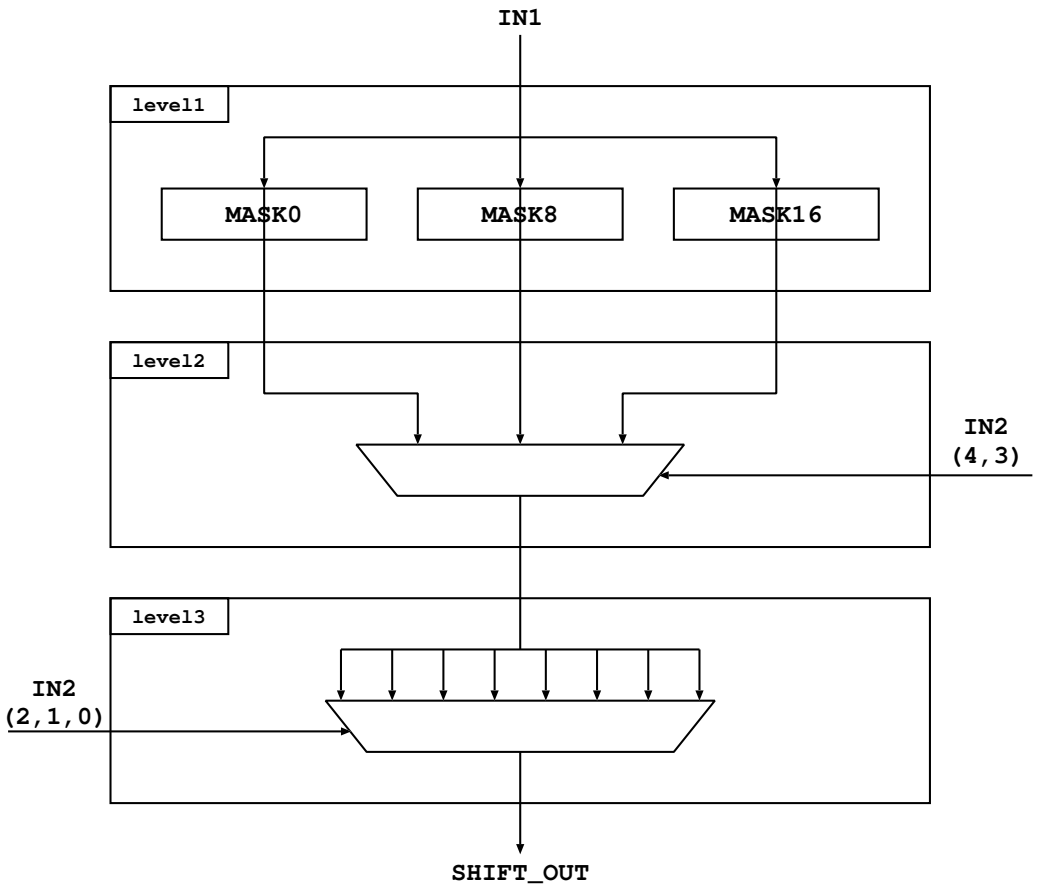

Figure 6: Shifter Block Schematic

- Level 3: The selection is done by the 3 LSBs of IN2. This last block computes a fine-grained shift which produces the final result by selecting the correct 32 bits among the 39 left. Please note that the values of IN2 are complemented in case of a right shift as we want an opposite behavior.

3.6 Memory Block

This block is responsible for all memory operations. Here we instantiate the data memory. It takes as input some data (from S register) and an address (from X register).

A multiplexer is needed to select whether the value passed to the next stage is the one coming from the memory or a bypass of the X value.

3.7 Forwarding Logic

FW_LOGIC is the block responsible of managing forwarding in the pipeline. It is able to detect RAW hazards and control forwarding multiplexers in decode and execute stage.
The computation is pretty simple, for each operation in EXE,MEM and WB stage we save whether is going to write in the register file (S_WB) and the destination register.
Possible Forwarding paths:

DEC Branches and register jump operations (JR,JALR,BNEQ,BEQZ) require the value of A in the decode stage. There are two paths, one from X register (Just after EXE stage), and from W register (Just after MEM stage). IR(26 downto 21 is) in compared with D2 and D3 to decide if forwarding is needed or not.

EXE Most R-TYPE and I-TYPE operations in the EXE stage. again we have two possible paths one from X reg and the other from W. I-TYPE operations require forwarding only for the A operand, thus only rA is compared, while R-TYPE require both operands so rB is compared too.

MEM A forwarding path in the MEM stage does not actually exist in our design because there is only one condition suitable from this purpose: when a store (SW) needs a register that is going to be written from a load (LW) in the previous clock cycle. We don't think this single case justifies a specific forwarding logic, so we avoided it. Please note that in any case the data taken is correct due to stall.

3.8 Control Unit

The control unit (CU) is in charge of generate and distribute control signal across the entire DLX pipeline. In this design, the control unit follows an Hardwired scheme, so at each IR input a kind of memory containing a control

work for each instruction is looked up. (There is actually no memory but an hardwired logic scheme)
When a signal needs to be used later in the execution, it is propagated trough a series of flip flops.
The CU is able to correctly generate and handle stall conditions due to hazards or multi cycle operations. In general, when a stage is stalled, the same goes fol the preceding stages in the pipeline, while the successive need to continue the execution. When a stage would be empty, a BUBBLE operation is inserted: a BUBBLE has the same control word as a NOP.

- **S_MUX_PC_BUS - (1,2)** : Control of PCMUX in the FETCH block
  - 00 - Register NPC4
  - 01 - Value read from A
  - 10 - Calculated jump address
  - 11 - Either A or jump address depending on BRANCHMUX
- **S_EXT - (3)** : Control of EXTENDER lenght in DEC stage
  - 0 - 16-bit extension
  - 1 - 26-bit extension
- **S_EXT_SIGN - (4)** : Control EXTENDER sign in DEC stage
  - 0 - Sign-extension
  - 1 - Zero-extension
- **S_EQ_NEQ - (5)** : Control of ZERO COMPARATOR in DEC stage
  - 0 - Output '1' when equal zero
  - 1 - Output '0' when not equal zero
- **S_MUX_LINK - (6)** : Control of LINK MUX in DEC stage
  - 0 - No link
  - 1 - NPC4 is bypassed to B

- **S_MUX_ALUIN - (7)** : Control of ALUMUX in EXE stage
  - 0 - IN2 taken from B register
  - 1 - IN2 taken from IMM register
- **S_MUX_DEST - (8,9)** : Control of DESTMUX in EXE stage
  - 00 - Unused
  - 01 - Destination take from rC
  - 10 - Destination taken from rB
  - 11 - Destination is R31
- **S_MEM_EN - (10)** : Memory enable in MEM stage
  - 0 - Disable Data Memory
  - 1 - Enable Data Memory
- **S_MEM_W_R - (11)** : Memory control in MEM stage
  - 0 - Read
  - 1 - Write
- **S_MUX_MEM - (12)** : Control of MEMMUX in the MEM stage
  - 0 - X bypass
  - 1 - Value taken from memory ouput
- **S_RF_W - (13)** : Write back enable in WB stage (sent to write port of REGFILE)
  - 0 - Disable write back
  - 1 - Enable write back

3.9 Stall Logic

Stall logic is a sub block inside the control unit in charge to distribute stall signal across the entire system.
The possible stall cases are:

- **DATA HAZARD**: Branches and register jump operations that need a forwarding from the previous operations: In case of a simple ALU operation, dec and fetch are stalled for a single cycle, in case of a load operation both are astalled for 2 clock cycles
- **DATA HAZARD**: All I-TYPE and R-TYPE operations preceded by a load (LW) instruction requires a single cycle sstall in fetch and decode stages
- **STRUCTURAL HAZARD**: When a multi cycle operation (MULT or MULTU) is in process, both fetch and decode stages are stalled
- **CONTROL**: After a misprediction, only dec stage is stalled

3.10 Instruction and Data Memories

As previously stated, both memories are not part of the design itself. Also, we will see that our design library is not suitable to synthesize such type of components.
For the previous reason, we did not put much effort on the memory design.

Instruction Memory Accessed during fetch operation. There is no enable port because we need to fetch an operation at each clock cycle. The memory is asynchronous.

Data Memory Accessed during MEM stage. There is an enable signal and a W/R. Both read and write operations are synchronous during the falling edge of the clock.

4 Simulation and Test

The simulation environment is composed as follows:
A test bench file tb_top_level instantiates the DLX together with the two memories and provides clock and reset signals to all the components.
Then there is a compiler Perl/Bash script that takes as input a DLX assembly code and compiles it into a memory dump ready to be loaded by both memories. This script, which has been provided us have a bug: when two or

more consecutive operations are exactly identical, only one is translated to binary code, while the other are simply empty lines.
We have also created a wave generation script (WAVE_BASIC.do) to be loaded directly into vsim that select the basic waveforms to be visualized.
Testing an entire processor is not an easy task, due to the very large possible combinations of instructions. On the very early stages of the design, testing was mainly done block by block in a bottom up fashion. for example, for the ALU, first we tested each single component such as adder and shifter, then we integrated them one at a time and checked it.
This approach has been used until we reached the simulation of the entire DLX block.
Then we moved into running test programs, starting from very basic ones to test each single operation, and then trying different combinations of hazards, forwarding, multi cycle operations and branch predictions.
For a final test, we ran a quicksort algorithm.

5 Synthesis

For the synthesis phase we used Design Vision tool from Synopsys, which is basically a graphical front-end for Design Compiler. The standard cell library provided us has reference gate size of 45nm.
This library, as previously mentioned, has a limitation: it is not suitable to properly implement large memories. For this reason, we decided to avoid synthesizing instruction and data memories, and to limit branch predictor size to 32 lines.
Synthesis has been widely used to evaluate critical paths all over the network, every time we did a modification on the design, we synthesized it to check for improvements. Because of this of this strategy we decided to:

- Move from a 13 to 14 cycles multiplication: in the first version of our multiplier we had a very long critical path flowing from the input to the accumulator because the multiplier was doing both encoding and an addition in the first cycle.

- Move Alu control signals generation to the decode stage: at the very beginning, ALU was receiving AluOP signal and had to decode it before the execution. This caused a delay before it could actually do the operation. Now everything is decoded in the decode stage and values

are save in a register in order to be available at the very beginning of the ALU execution.

- Move Register file: Register file was initially reading the value during the falling edge in the middle of the decode stage. This caused the required time starting from regfile to be half of the period, causing a strong impact on critical path. Now the read operation is done at the beginning of the decode stage.

- Change the structure of the microcode memory: synthesizing the control unit gave strange results at the beginning because of array usage(we still don't know why). Moving it another block and using the case statement fixed this unwanted behavior.

The final TCL synthesis script can be found in the project folder. After some trial and error, we found a good way to have the best results from the tool. Execute block is characterized and synthesized alone because we think it is the critical zone in the design, both in execution and decode stages. The minimum required time for this block can be pushed down to 1.05ns, but we left some margin and set it to 1.2ns Done with the execute block, we set it as don't touch and proceed to compile the entire processor. Again, we set a safe minimum clock period to 1.6 ns (625 MHz clock frequency).
It's interesting to see from the final reports that the critical path does not flow through the ALU, instead it goes in the decode stage and after in the branch predictor.
All the compile commands are done with compile_ultra command and the -no_autoungroup parameter. The no_autoungroup gives a little worse results but at least preserves information regarding critical paths. compile_ultra is also able to do some smart optimizations such as cutting the adder in the decode stage from 32 to 26 bits and complement some signals to have a better mapping results.

5.1 Power Optimization

We decided to evaluate the impact of clock gating technique on our design. To do so, we followed a set of commands we found on the design_vision manual.

```
elaborate TOP_LEVEL -architecture ARCH -library
   DEFAULT
set_clock_gating_style -sequential latch
create_clock -name "clk" -period 1.6 clock
insert_clock_gating
propagate_constraints -gate_clock
compile_ultra -no_autoungroup
```

This reduced the overall dynamic power consumption by approximately 27% as shown in Table 3.

Design	Total Dynamic Power [mW]
clk@1.6ns	12.0339
clk@1.6ns clock_gating	8.7435

Table 3: DLX - Clock Gating Power Reduction Evaluation

It's also interesting to see the trade-off curve while changing the maximum clock frequency as shown in Figure 9.

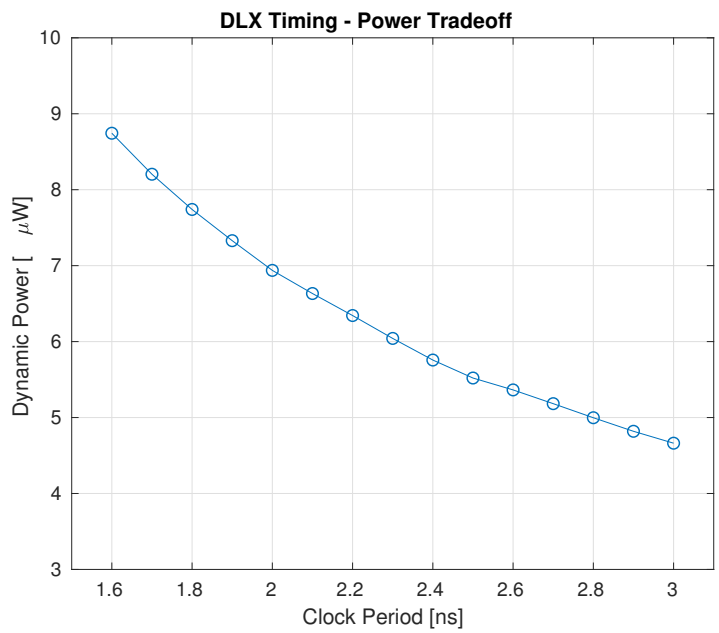

Figure 7: DLX - Timing/Power Tradeoff Curve - Clock Gating Enabled

6 Layout

Finally as our last step, Physical design of the entire processor has been done. We followed the same procedure as the one described in the laboratory instruction, because we don't know the Encounter tool very well.
We decided to do the Floorplanning only for the non-clock-gated design with clock period of 1.6 ns.
Due to the large size of the design, some processes took a lot of time, for example Clock Tree Synthesis and Cell Placement.
At the end we printed all the reports, luckily our design show no geometry or connectivity violations. Only some hold timing violations appear to be present.
Among all the reports, we extracted some values regarding gate count and area: It's interesting to see that more than 60% of the overall area is occupied by BTB and Register File.

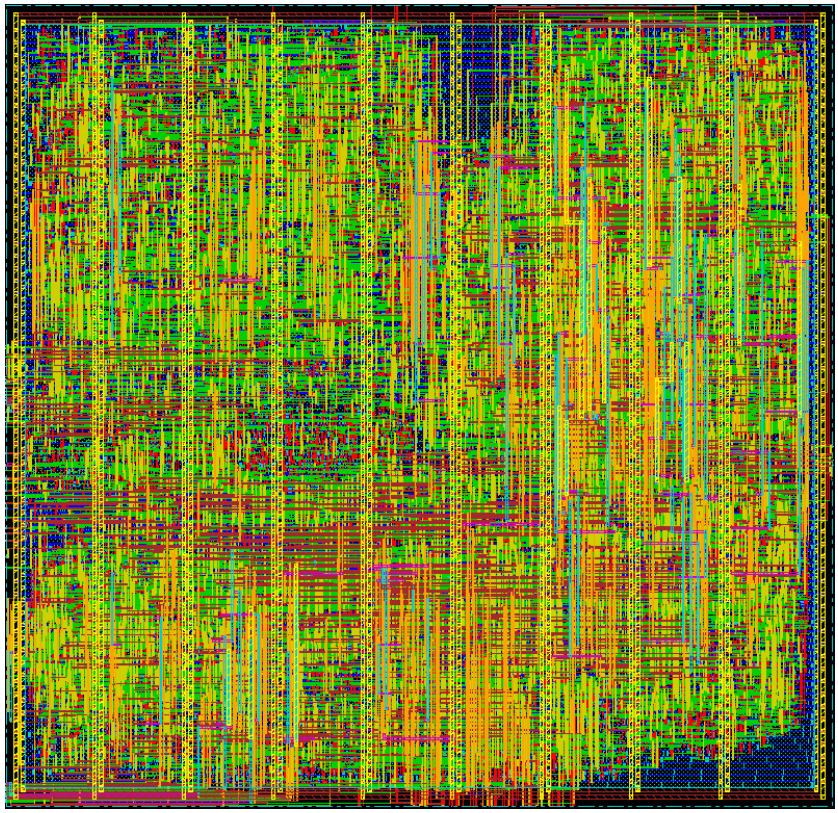

Figure 8: DLX - Post Place&Route Schematic from Encounter

Block	Gate Count	Area(μm^2)	%
Total	26524	21166.7	100%
Register File	10444	8334.3	39%
BTB	6401	5108.0	24%
ALU	4088	3262.2	15%

Table 4: DLX - Post Place&Route Gate Count and Area

7 Conclusions

This project required a incredible amount of work and taught us that microelectronic design is not so simple.
As a future work, this design could be extended by including the missing instructions and adding the Floating Point part of the microprocessor. Also it would be interesting to see some numbers regarding the branch predictor, in particular a tradeoff curve between size and misprediction rate with real life benchmarks.

References

[1] *DLX Instruction Set*. URL: `http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/DLXinstrSet.html`.

[2] Mariagrazia Graziano. *Microelectornic Systems Lecture Notes*. 2013.

[3] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN: 1-55880-069-8.

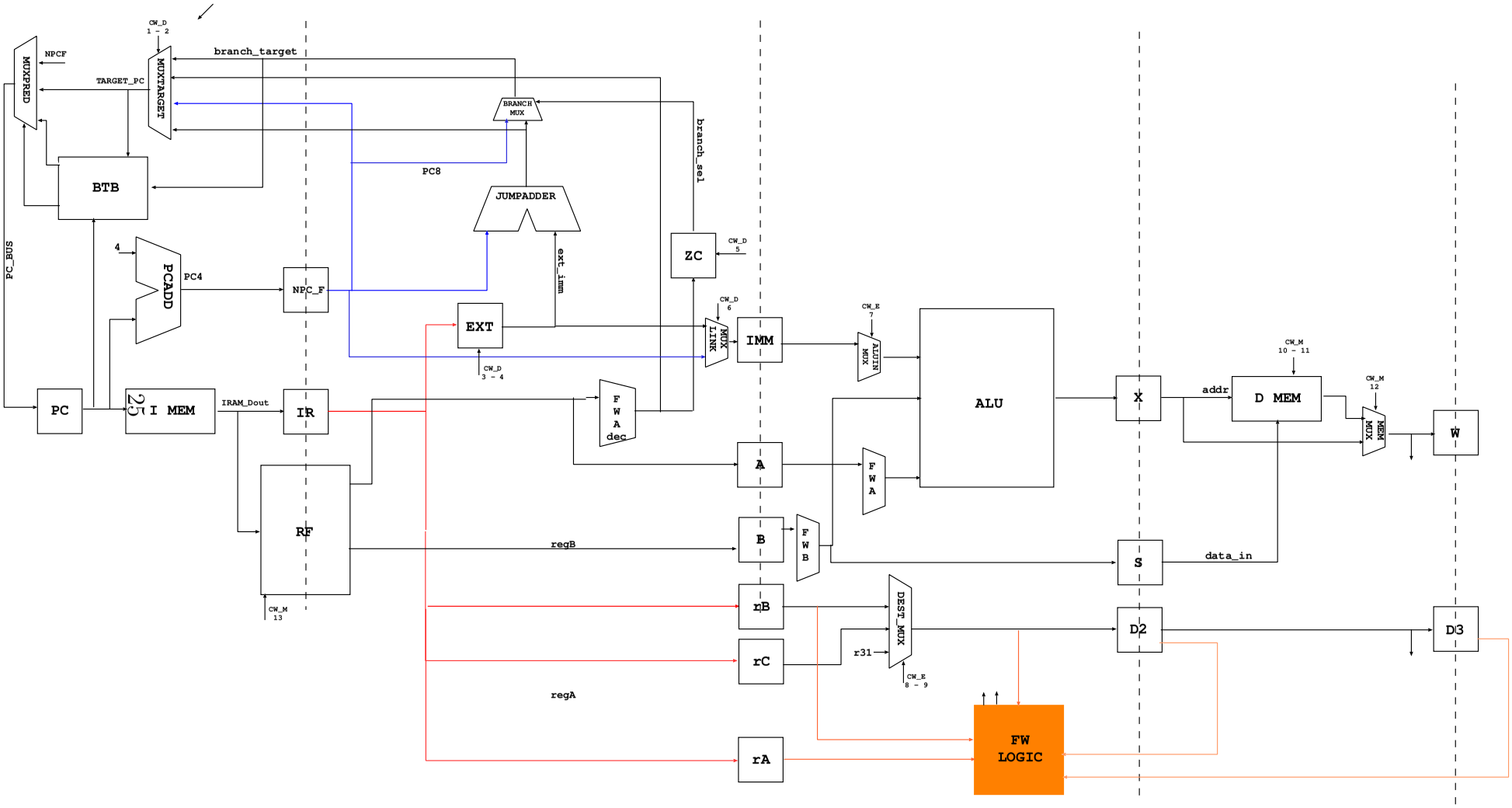

Figure 9: DLX - Full Datapath Scheme