DLX

October 24, 2017

1 Introduction

The goal of this project is to implement a completely functional DLX processor in RTL 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 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.

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 3 and 6 are about the environment and results of design and layout phases. Finally, Section 4 gives the conclusions and possible future work.

2 Implmentation Overview

DLX is a classical 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 BLABLALBLALBLAA.

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 13 cycles (9 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 is sent from the control unit.

Memories   Two separated memories for instruction and data have been included in the design, but they're 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 or 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. Due to the position of the register file and the moment in which the control world is computed, it is not possible to selectively enable or disable read ports because it is not known a priori whether the instruction needs one or two source register.

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
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 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 not sent to decode stage, and it is replaced by the correct one on the successive clock cycle.

3.2   Branch Predictor

As previously stated, a branch predictor scheme have 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 deadlock conditions.
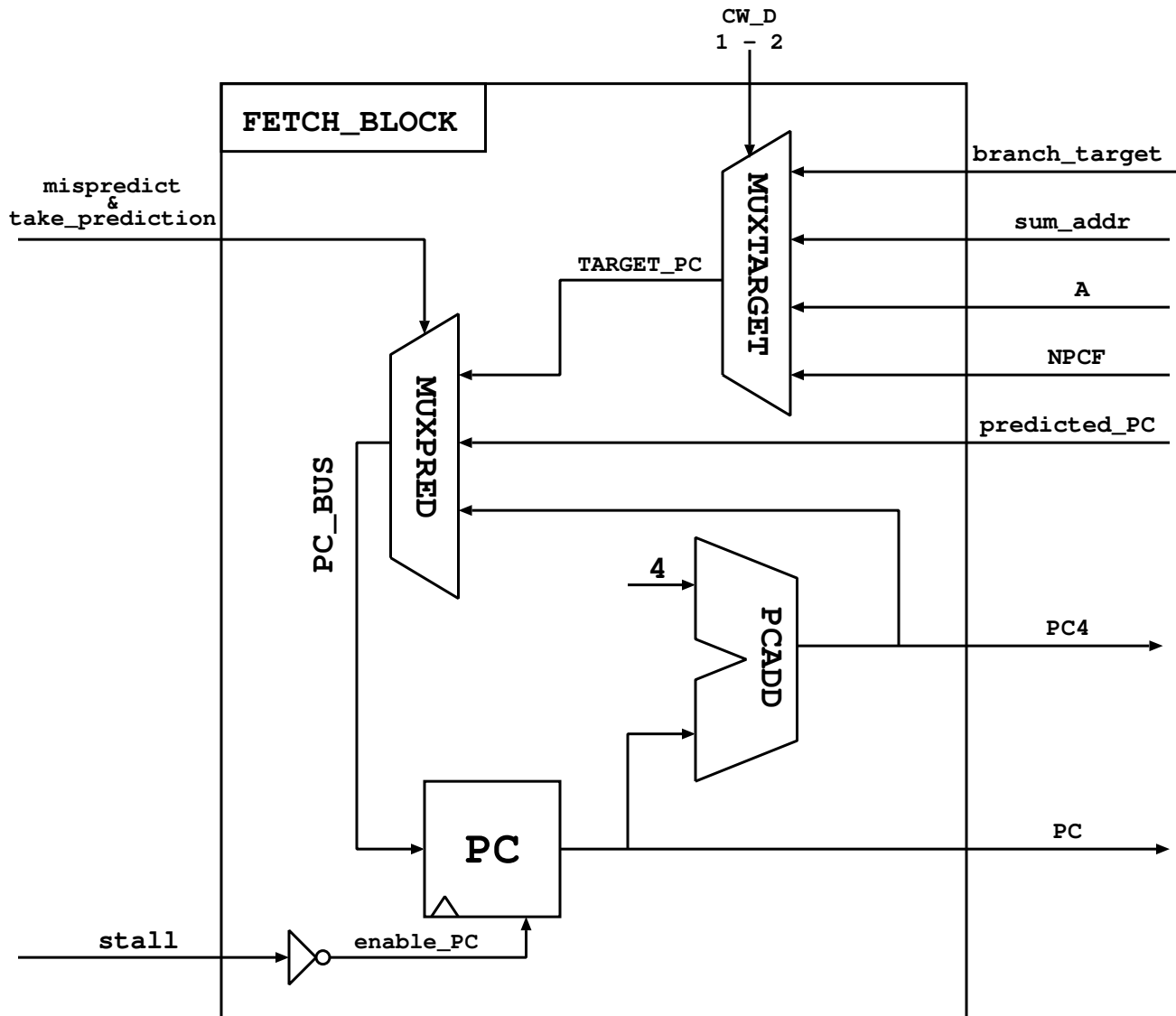
Figure 1: FETCHBLOCK

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.
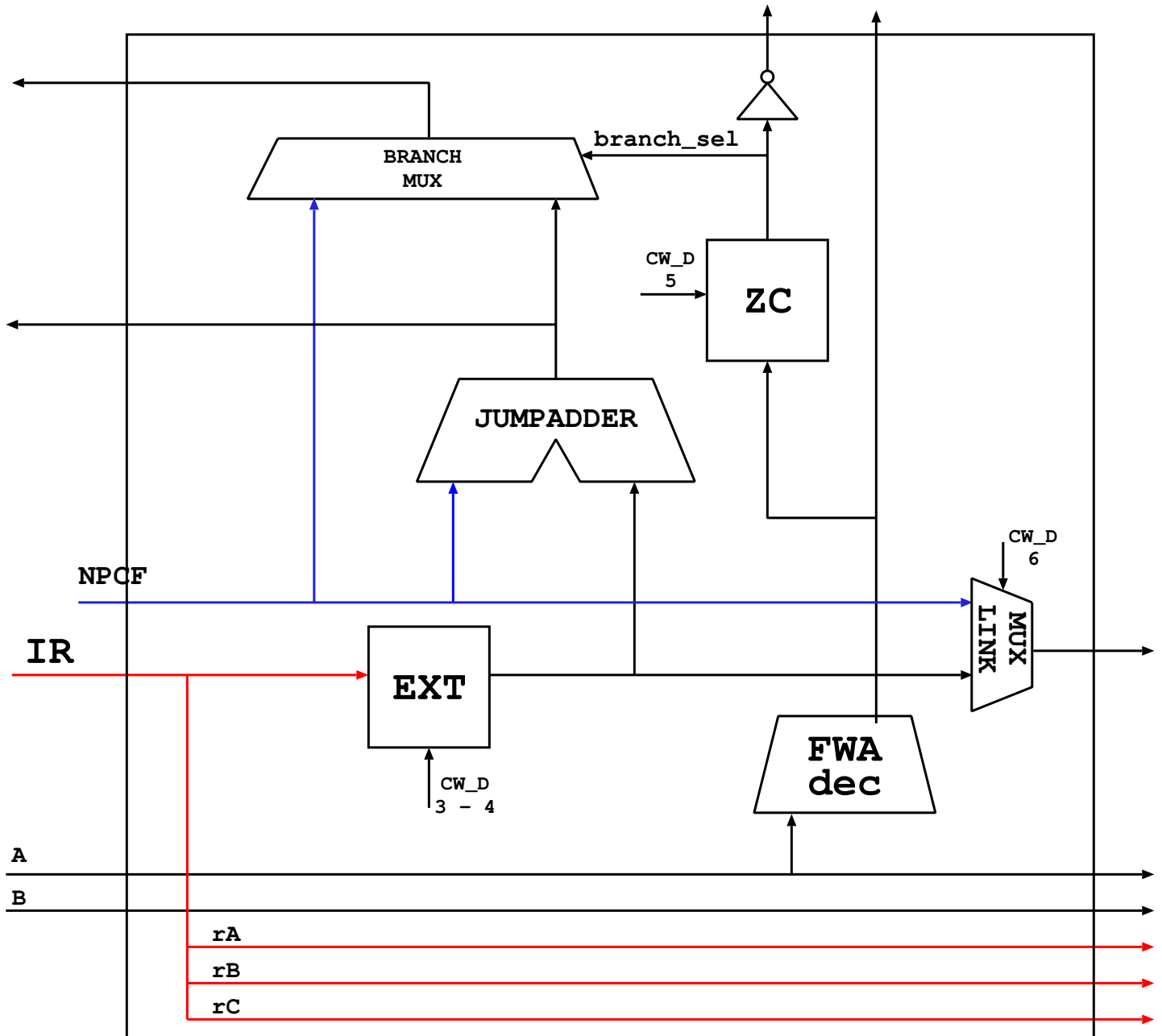
3.3   Decode Block

Figure 2: DECODEBLOCK

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 forget about forwarding multiplexers, as they are going to be described later.
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.

3.5   ALU and Multiplier

The ALU is responsible for all data manipulation in our processor.
It takes as input two 32 bit operands and a special code that specify the type of operation that is going to be performed. 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.

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.
The adder is composed of two main blocks: 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.
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 developed. From basic computer arithmetic we know that doing a subtractions is the same as doing an addition with the second operand being 2's complemented. 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.

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. 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.

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.

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.

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.

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

There might be some redundancy in the design, we left possible optimizations to the synthesizer.

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 (mask00), 8 (mask08) and 16 bits (mask16) each. Moreover, operand A 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 B does the selection of one of the masks produced in the first block.

- Level 3: The selection is done by the 3 LSBs of B. 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 B are complemented in case of a right shift as we want an opposite behavior.

3.6   Memory Block

NOTHING MUCH TO SAY: ADDRESS ALWAYS COMES FROM ALU, DATA ALWAYS FROM S MUX TO IGNORE MEMORY NOT ENABLED WHEN NOT NEEDED

3.7   Forwarding Logic

Forwarding 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.

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)

S_EXT - (3)

S_EXT_SIGN - (4)

S_EQ_NEQ - (5)

S_MUX_LINK - (6)

S_MUX_ALUIN - (7)

S_MUX_DEST - (8,9)

S_MUX_MEM - (12)

S_MEM_W_R - (11)

S_MEM_EN - (10)

S_RF_W - (13)

3.9   Stall Logic

LIST STALL CASES (HAZARDS) Control Hazards: MISPREDICTION Data hazards: RAW only: MEM/EXE solved by FW WB/EXE solved by FW DEC/EXE solved by stall: Branch or Jr situation
STALL COMING FROM EXECUTE Structural hazard: Multiplication multi cycle operation

3.10   Instruction and Data Memories

INSTRUCTION AND DATA MEMORIES NOT PART OF THE DESIGN ITSELF, SIZE OF DMEM AND IMEM As previously stated, both memories are not part of the design itself. Imem is accessed during fetch operation, no enable port because we need to fetch an operation at each clock cycle. Dmem accesses during MEM stage. There is an enable signal and a W/R. both read and write are synchronous during the falling edge of the clock. to move it at the rising edge need to move the control signal before, so new values are ready at the clock edge.

4   Simulation and Test

The simulation environment is composed as follows: A test bench file: tb_top_level instantiates the dlx and 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 (WABE_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 comparator and shifter, then we integrated them one at a time and checked it.
An approach like this has been used until we reached 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

Not simple. separated CU and datapath. ST 65nm library. not suitable to synthesized memories, problems on BTB due to high number of lines. reduced to 32 lines for synthesis. all-flatten and compile ultra option gives extraordinary results but doesnt give informations on critical path.

6 Layout

In this section we describe the results.

References