DLX Project

Pane Alexander James - 221919
Fioretti Yuri - 211386

October 28, 2015

Contents

Summary

This projects consists in the developing in VHDL code, testing, synthesize and design the physical layout of a DLX processor. The final version is derived from the development of a basic version of the DLX, where various features have been added to increase computational performance and to avoid various hazards.
For developing the basic version an architecture has been designed as follows:

- 5 stage pipeline implementation: both Datapath and Control Unit are design implementing a pipeline architecture. As it will be explained after the Control Unit is FSM based.
- The instruction set that can be executed is:

  R-Type instructions: add, and, or, sge, sle, sll, sne, srl, sub, xor.

  I-Type instructions: addi, andi, ori, sgei, slei, slli, snei, srli, subi, xori.

  B-Type instructions: beqz, bnez.

  J-Type instructions: j, jal.

  L-Type instructions: lw.

  N-Type instructions: nop.

  S-Type instructions: sw.
- A Datapath that could fulfil all the instructions listed above

After developing a fully functional basic DLX we passed to a PRO version. This versions presents new features:

- An advanced instruction set that expands the previous one with the following instructions:

  R-Type instructions: addu, seq, sgeu, sgt, sgtu, sleu, slt, sltu, sra, subu.

  I-Type instructions: addui, lhi, seqi, sgeui, sgti, sgtui, sleui, slti, sltui, srai, subui.

  J-Type instructions: jalr, jr.

  L-Type instructions: lb, lbu, li, lhu.

S-Type instructions: sb, sh.

F-Type instructions: div, divi, mult, multi.

- Implementation of a divisor and of a multiplier. For this last one, since it has a long latency, a 2 stage pipeline version has been implemented.
- Implementation of a Hazard Unit for solving issues of multicycle operations, data hazards and control hazards:

  for multicycle operations the Hazard unit sends a specific signal.

  for data hazards a forwarding unit has been implemented that also includes the insertion of stalls.

  for control hazards a static prediction was used, but afterwards a BPU has been implemented.
- for the instructions a L1 cache has been implemented.
- to check the functionality of the architecture various benchmarks have been used.
- The processor has been synthesized.
- The physical layout of the processor is placed and routed.

Chapter 1

Datapath

The Datapath is where all the data flows to generate at the end the result of the operation and store it in the memory or in the registers. It's composed of 5 stages, each divided by a register (pipeline), and the components are controlled from the Control Unit that generates a control word for each instruction, making the data flow in the right direction.
The Datapath readd and/or writes the data from/to the registers and the memory. In the basic version both memories are necessary, the instruction memory and the data memory, and they are allocated outside the processor; while in the PRO version a cache has been implemented for the instructions leaving only the data memory outside the processor.
All the instruction set of the basic DLX can be executed in maximum 5 clock cycles, 1 for each stage, while for the PRO version multicycle instructions have been implemented.

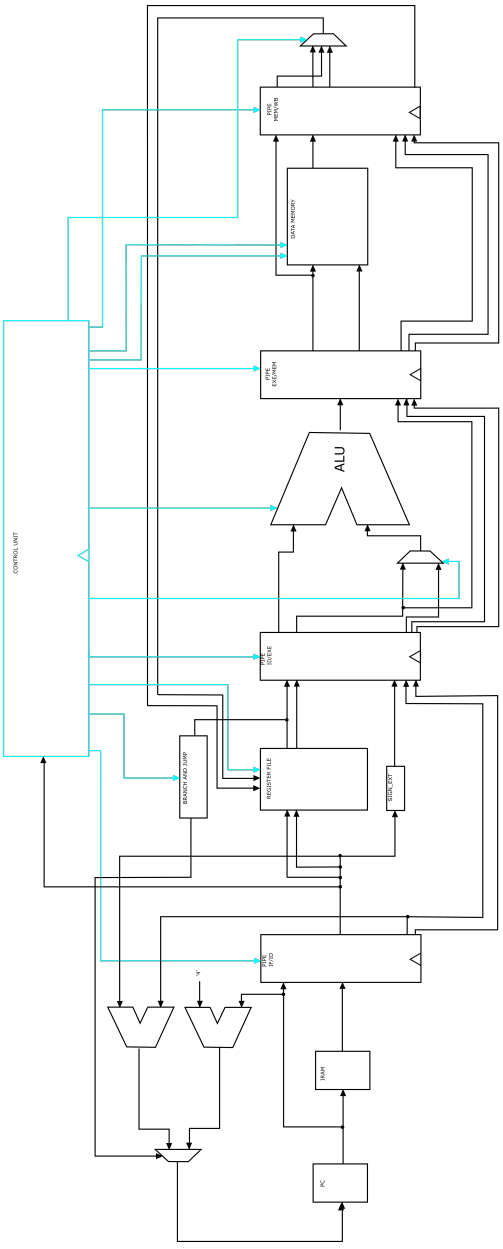


Figure 1.1: Datapath of the basic version of the DLX

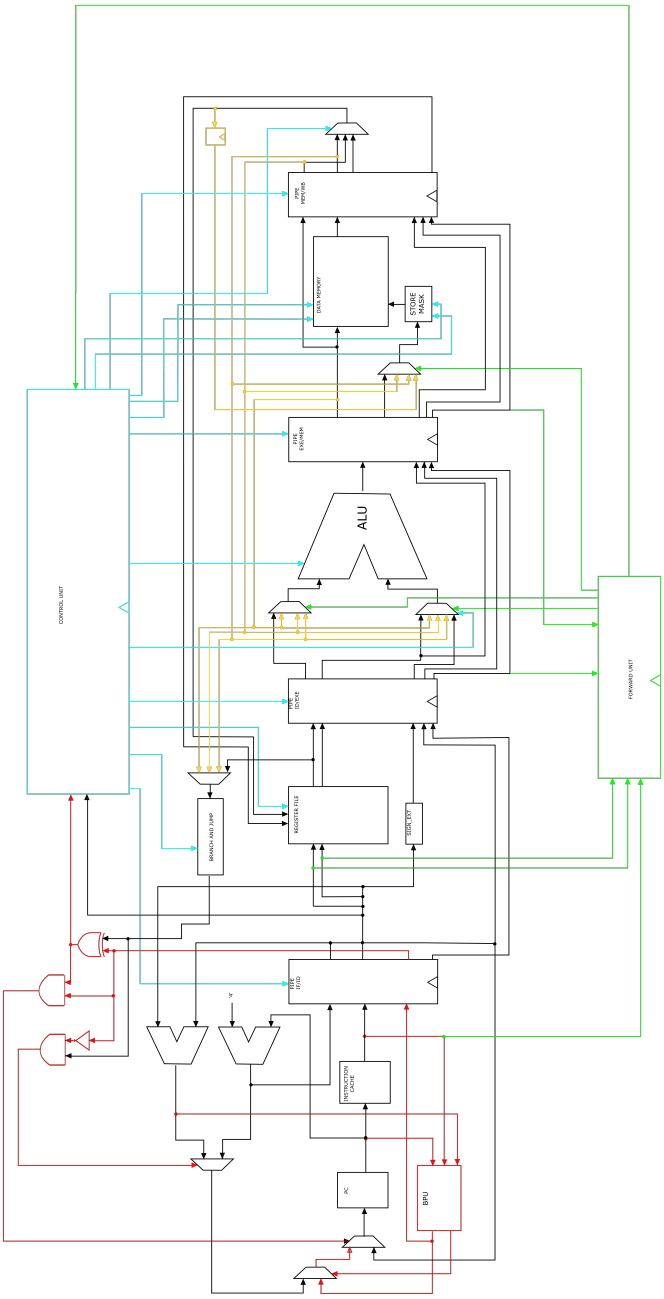


Figure 1.2: Datapath of the PRO version of the DLX

1.1 Fetch

In this stage the register named program counter points in the instruction memory the next instruction to be fed to the Datapath, as explained before the instructions of the basic microprocessor are gathered from the memory allocated outside, while in the PRO version the instructions will be allocated in the instruction cache. During this stage also the OPCODE and the FUNC is fed to the Control Unit.
To retrieve the next instruction the program counter is increased each time by four, unless a jump or a branch instruction is present, in this case the program counter will change depending on the data present while computing the branch. For this purpose a Multiplexer is present at the input of the program counter that will select if the next instruction will be the contiguous one or an instructions elsewhere.
The result of the branch is computed during the Decode stage so the program counter, in case of a jump or a taken branch, could assume a wrong value and the instructions doesn't need to be executed and a clock cycle will be lost. This problem is known as Stuctural hazard is also been taken into account and will be discussed later.

1.2 Decode

In this stage the instruction is split in various parts and depending on the typology of the instruction different ranges are taken into account. It's possible to distinguish two main kinds of instructions: R-TYPE and I-TYPE.

R-TYPE:

OPCODE (6 bit)	RS1 (5 bit)	RS2 (5 bit)	RD (5 bit)	FUNC (11 bit)

For this kind of instruction the OPCODE is fixed to "000000", so that the Control Unit is able to distinguish that it's an R-TYPE, while the FUNC field contains the details on the kind of instruction, so both this values are taken into account from the Control Unit.
The fields RS1 and RS2 represent the two registers to be read, while the field RD represents the destination register of the operation.

I-TYPE:

OPCODE (6 bit)	RS1 (5 bit)	RD (5 bit)	IMMEDIATE (16 bit)

For this kind of instruction the OPCODE contains the details on the kind of instruction.
The fields RS1 represent the register to be read, the field RD represents the destination register of the operation, while the field IMMEDIATE represents the second value to use for the operation.
The Control Unit will manage the multiplexer and the register file through the Control Word to perform the correct operations.

1.2.1 Branch prediction

Another problem to keep into account during an execution of a program is branching. Since it's not possible to know during the Fetch stage if the branch is taken or not, because a register needs to de read, so only in the Decode phase it's possible to know the result of the branch and then use the ALU to compute the right address, thus causing two possible misreads instructions and so two clock cycles would be lost.
But since another adder, connected to the previous program counter and to the immediate value of the instruction, has been placed in the decode stage for jumping and branching, so it's possible to have the result in this stage and thus fetching only one wrong instruction and so losing only one clock cycle, if the branch is taken or if it was a jump.
To know the result of the branch the following circuit has been implemented:

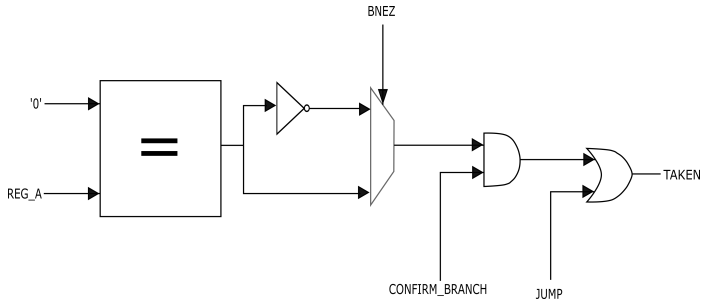


Figure 1.3: Component to compute the result of the Branch

But since the instruction would in any case be fetched we can assume that this architecture implements a static prediction for branches, since if the result of the branch is not taken no clock cycles are lost. So all jump instructions (since there are always taken for definition) will always fetch a wrong instruction and so lose one clock cycle.

1.3 Execute

The Execution stage is where the main computation on the variables is executed by the ALU (Arithmetic Logic Unit) where all the dedicated hardware to solve

the operations is allocated. A multiplexer is present on the second input of the ALU to choose the right value to feed to the ALU, since it can be an immediate value or the value of a register.
Inside the ALU is composed these four components:

- Pentium 4 adder: it's the main component that computes additions and subtractions. It's a kind of tree added that computed the carry each four bits, to be precise the tree is called Sparse-Tree.
- T2 logicals: it's the hardware that computes the logicals operations like AND, OR, XOR, etc. All the logic operations are done Bitwise.
- Comparator: this hardware will produce the results of various comparations, like equality, majority or minority.
- T2 shifter: this hardware performs the logical or arithmetical shift of a register.

Also with the PRO version of the device other hardware has been implemented in the ALU:

- Booth multiplier: a three stage pipeline multiplier based on the Booth algorithm has been implemented.
- non restoring divider: a divider based on the non-restoring division operation has been implemented, taking 18 clock cycles to finish the operation in the ALU.

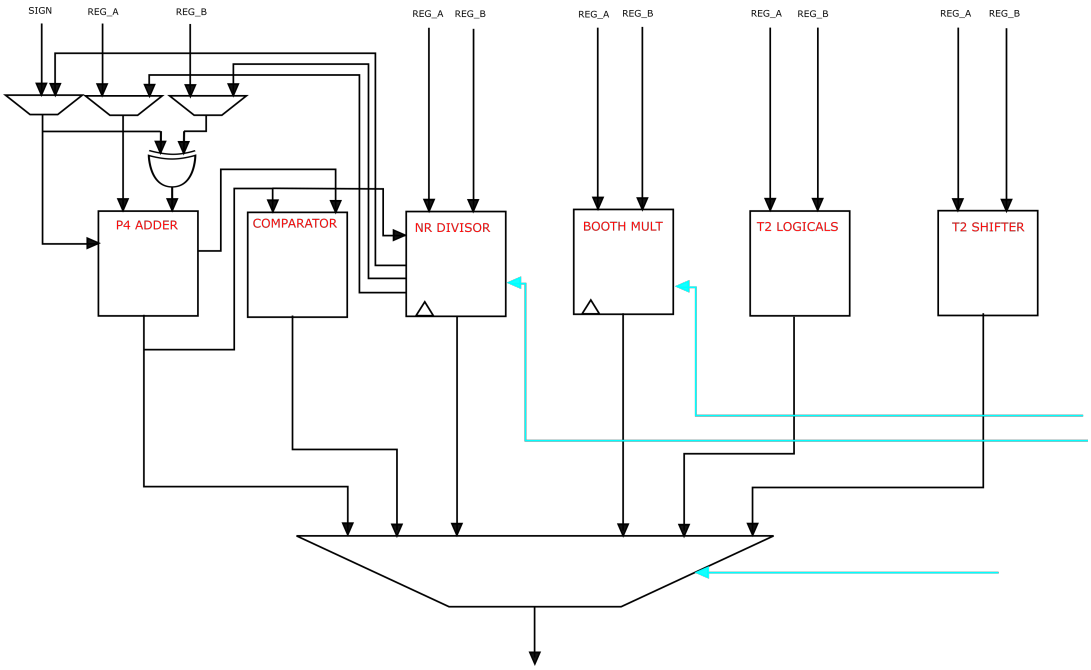


Figure 1.4: ALU

The Pentium 4 adder has been already been developed during the lab sessions, where it's structure has been descripted in a totally structural way. The adder is composed of two sections: the carry look-ahead, that generates the carry produced each 4 bits, and the actual adder which is composed of eight four-bit carry select adders (CSA) working in parallel. The adder can also perform subtraction, since a XOR gate on the second input of the adder has been allocated, which performs the bitwise logic operation between the SUB/ADD signal and the vector, generating so the bitwise negation of the vector in case of subtraction. Then the SUB/ADD signal is used as carry in of the adder performing so the addition between the first input and the 2'complement of the second input, which means the subtraction between first and second input.

The T2 logical is composed, like it has been explained during class, of five NAND gates. Through De Morgan laws it's possible to perform all the logical operations required by the instruction set.

The comparator is a block that, to perform its task, needs to read the result produced form the Pentium 4 adder and it will execute the NOR logical operation between all the bits of the result given, so it will produce al logic '1' if all the bits are '0' out of the adder, an it will produce a logic '0' if at least one bit is '1'. It's so possible to generate all comparations needed.

The T2 shifter has been implemented in a totally structural way, using the three stages as explained during the class, that means:

- generating all the masks through multiplexers.
- performing a course grain shift from the 2 highest bits of the lower 5 bits of the second operand.
- performing a fine grain shift from the 3 lower bits of the second operand.

The hardware can perform logic shifts right or left and arithmetic right shifts.

The multiplier, as we said, is based on the Booth algorithm, this component has been descripted already during the lab sessions. But in the DLX a slightly different version has been implemented, which doesn't use RCA with an increasing number of bits, but all of them perform a constant 35-bit addition , also this hardware has been pipelined, this will be explained in the next chapters.

The non restoring divider, like the comparator, to achieve its task it must communicate with the Pentium 4 adder, since at each clock cycle it will perform a subtraction or an addition of the residue at the i-th clock cycle.
This hardware requires 23 clock cycles to complete the operation but it's possible to perform division even with negative operands.

1.4 Memory

This stage is mostly used for load and store instructions, here is where the memory can be accessed to be read or written. The memory has two inputs, one for the address and the other one for the data to be store, and one output, which will produce in case of a load isntruction the data out of the memory.
In the PRO version an AND mask has been introduced at the input data of the memory to perform the SB and the SH instructions, so in case of a LW instructions the data is masked with all ones, so the data remains unchanged. In case of a LH instruction the mask will AND with zeros the upper 16 bits. And in case of a SB instruction the upper 24 bits are masked woth zeros.
Also on the output of the memory a five to one multiplexer is present, this to implement the LH, LHU, LB, LBU instructions.

1.5 Writeback

This stage has the function of selecting the correct data to be written in the register file, since the possible data could be the data from the memory or the data from the ALU. Also in this stage the PC, relative to the instruction in WB, added by four is present since the JAL and the JALR instructions require to store in the register file the return address.
In this register is also present another register, this is to perform forwarding towards the memory in case of store instructions.

Chapter 2

Control Unit

The Control Unit is the main hardware that controls all the control signals of the various registers, multiplexers, etc. To be adapted to the datapath the Control Word has been pipelined, where in the first stage both the Fetch and the Decode signals are given together, then all the other stages pass through registers.
Advancing to the PRO version other signals have been added to the Control Word to be able to manage the increasing hardware, and also other signals need to be read from the Datapath to avoid various problems (stalls, wrong fetches, etc.).

2.1 The FSM implementation

For the Control Unit it has been chosen to use an FSM implementation. The data that need to be read to change to the next state are the OPCODE and the FUNC, which are retrieved from the Fetch stage, so depending on these values the state of the machine will change, so the current state of the machine represents the instruction that at the moment is present in the Decode stage.
As we said before other signals have been added in the PRO so other states have been added, like the LOAD state for filling the cache entry.
The image below shows indicatively the behavior of the FSM:

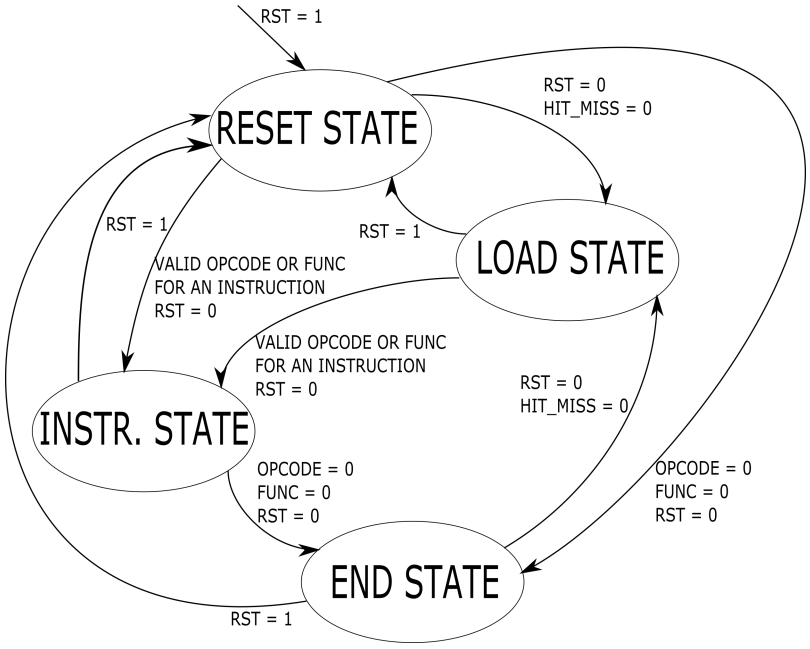


Figure 2.1: Flow chart of the Control Unit

2.2 Stalls due to Hazard unit

As explained before the PRO version has a Hazard unit implemented, but some hazards need to stop the pipeline's flow of data, since the data needed is not yet produced. So also the Control Unit must aknowledge this stall of data and stop the Fetch, the Decode and the execute pipelines, so that the instruction that needs the data will be blocked in the Execute stage. It also can be possible that stalls are generated from branch instructions, so in this case only the Fetch and the Decode stage must be stalled.
When the data will be ready (most of the stalls in this case last for just one clock cycle) it will be forwarded to the Execute stage and the pipes will be enabled again, producing so the correct result, or will be forwarded to the Decode stage in case of branches.

2.3 Stalls due to Cache misses

After implementing the Hazard unit a first level L1 cache for the instructions has been inserted. So since we are dealing with a memory that might need to gather the data from the main RAM memory, so when a miss event occurs, the Control Unit must then stop the normal flow of data giving the right time to the cache to fill a new array of instructions.

To achieve this goal the Hit/Miss signal is arised and the Control Unit by reading this signal aknowledges that the cache needs refilling, and will not fetch new instructions, but will complete the ones already flowing in the Datapath.
The time necessary to the cache to perform this operation is scanned from a five bit synchronous counter that is present in the cache module itself, so after refilling the array the Hit/Miss signal will be driven low and the Control Unit will stop stalling the flow of data.

Chapter 3

The hazard unit

3.1 Data hazards

Since some kinds of instructions need to store the result of the operation in the register file, it can be possible that one of the next instructions requires a data that is been manipulated and not saved yet. These kind of dependencies are called Data hazards, and they can be classified in four case: write-after-read (WAR), write-after-write (WAW), read-after-read (RAR) and read-after-write (RAW).
but since there isn't any problem if we want to read a data that another instruction just read and didn't modify it (RAR and WAR cases), and since in a WAW the registers required in common are the destination registers, while the data that needs to be read was left untouched. The true dependencie is only in RAW cases.
Some cases are illustrated below:

- The instruction requires, in the EXE stage, a data that is being changed in the instruction before.

INSTRUCTION	1	2	3	4	5	6
ADD R1, R2, R3	IF	ID	EXE	MEM	WB	-
ADD R4, R5, R1	-	IF	ID	EXE	MEM	WB

- Three instructions are consecutively dependent one from the next one.

INSTRUCTION	1	2	3	4	5	6	7
ADD R1, R2, R3	IF	ID	EXE	MEM	WB	-	-
ADD R4, R5, R1	-	IF	ID	EXE	MEM	WB	-
ADD R6, R1, R4	-	-	IF	ID	EXE	MEM	WB

- The instruction requires data that needs to be loaded, from the memory, from the previous instruction.

INSTRUCTION	1	2	3	4	5	6
LW R1, 10(R2)	IF	ID	EXE	MEM	WB	-
ADD R3, R4, R1	-	IF	ID	EXE	MEM	WB

- The data required from the instruction is needed at the MEM stage, but it's being modified from the previous instruction.

INSTRUCTION	1	2	3	4	5	6
ADD R1, R2, R3	IF	ID	EXE	MEM	WB	-
SW 15(R4), R1	-	IF	ID	EXE	MEM	WB

- The data required from the instruction is needed in the ID stage, but it's being modified from the previous instruction in the EXE stage.

INSTRUCTION	1	2	3	4	5	6
SUBI R1, R0, 1	IF	ID	EXE	MEM	WB	-
BNEZ R1, LABEL	-	IF	ID	EXE	MEM	WB

- The data required from the instruction is needed in the ID stage, but it's beeing modified from the previous instruction in the MEM stage.

INSTRUCTION	1	2	3	4	5	6
LW R1, 1(R0)	IF	ID	EXE	MEM	WB	-
BNEZ R1, LABEL	-	IF	ID	EXE	MEM	WB

3.1.1 Solution adopted for data hazards

- For solving the first two kinds of hazards a simple forwarding from the MEM stage towards the EXE stage has been implemented.

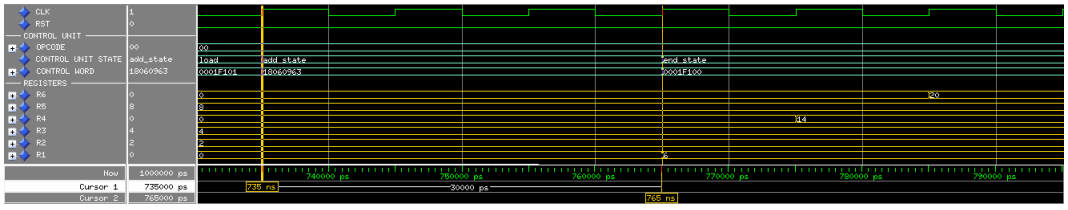


Figure 3.1: CU waves and relative results stored in the registers

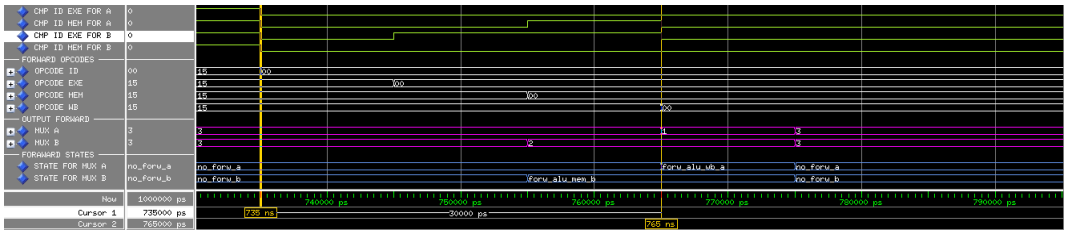


Figure 3.2: general waves, states assumed and relative outputs of the Forwarding unit

In **Figure 3.1** it's possible to see the states assumed from the CU (the change is relative to the intruction in the DECODE stage), the Control word generated and the results in the registers, which are correct even if they had dependencies
In **Figure 3.2** the main waves of interest of the farwaring unit are displayed,. The CMP ID signals are the results of the comparators between the DECODE stage and the EXECUTE and MEMORY stages. When the second ADD instruction is in DECODE the first one will be in EXECUTE, so the comparator wiil switch to '1'. At the next clock cycle the second ADD will be in EXECUTE and the required data will be forwarded, in fact it's possible to see that the state changes in FORW_ALU_MEM_B (B is for the bottom mux of the ALU), and the MUX B signal will then drive the correct data in the ALU.
At the same time the third ADD will be in DECODE, so the values of the comparators will change, infact this time there will be correspondence between ID-EXE and ID-MEM. So the FSM for MUX A will need to forward the data from the WB, while the FSM for MUX B will forward the data from the MEM stage.

- To solve the third kind of hazard a simple forwarding isn't enough, since we must wait that the LOAD instruction reaches the WB state while the next instruction stage remains in the EXE stage, so a stall has been implemented. After the stall the data will be forwarded from the WB stage to the EXE stage.

INSTRUCTION	1	2	3	4	5	6	7
LW R1, 10(R2)	IF	ID	EXE	MEM	WB	-	-
ADD R3, R4, R1	-	IF	ID	S	EXE	MEM	WB

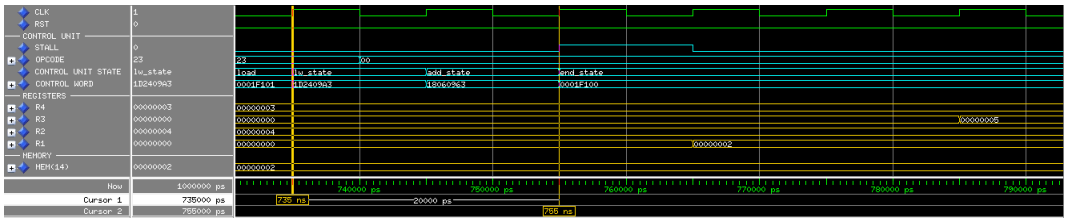


Figure 3.3: CU waves and relative results stored in the registers and in the memory

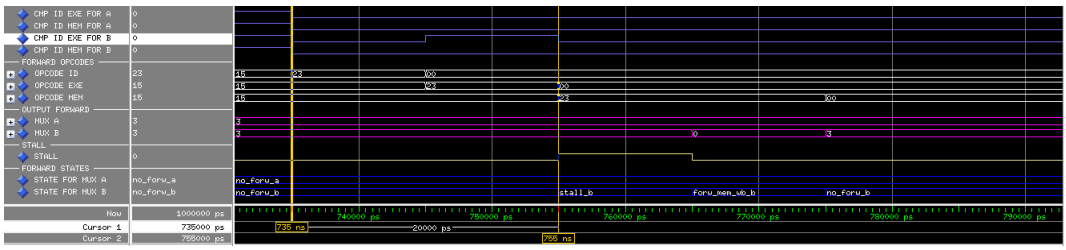


Figure 3.4: general waves, stall wave, states assumed and relative outputs of the Forwarding unit

In **Figure 3.3** we can see that the result is correct, but one clock cycle later the result is stored in the register, infact we can see that the stall signal in CU was high for a clock period, giving the time to the load instruction to pass in WB and forward the required data.
In **Figure 3.4** it's possible to notice that the forward unit passes in a STALL state, setting to '1' the STALL signal so that the CU will acknowledge and stall the pipeline after the EXE stage.

- For the fourth topology the STORE instruction has gathered the wrong value in the ID stage, so here a simple forwarding needs to be done, this time from the WB stage to the MEM stage

- For the fifth kind the BRANCH instruction needs the data at the ID stage, since as we said before the addition of the new address is performed in this stage, but the instruction before has changed the value and must wait a clock cycle so that the data has been computed, it's a similar case to the third one. So a stall is introduced to keep the branch in the ID stage so that the forwarding can be performed from the MEM stage.

INSTRUCTION	1	2	3	4	5	6	7
SUBI R1, R0, 1	IF	ID	EXE	MEM	WB	-	-
BNEZ R1, LABEL	-	IF	S	ID	EXE	MEM	WB

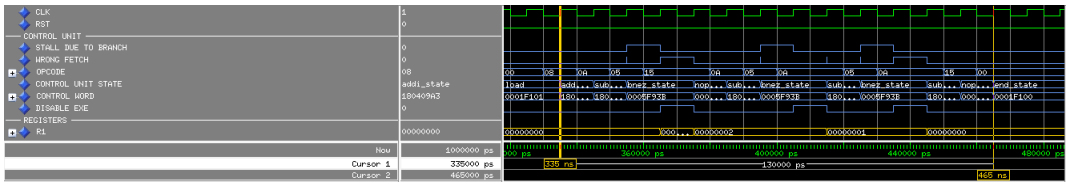


Figure 3.5: CU waves, stall and wrong fetch signals and relative result stored in the

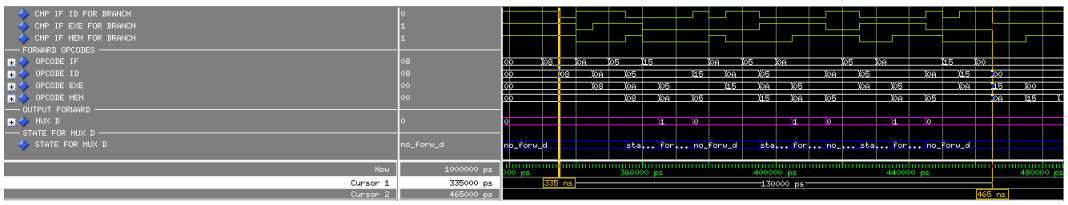


Figure 3.6: general waves, stall wave, stats assumed and relative output of the Forwarding unit

In **Figure 3.5** we can see that the result is correct, first of all we add to R1 to the value of the loop, in this case three, and the branch instruction will remain for two clock cycles in DECODE stage, since the data modified from the subtraction isn't ready yet. The first time the instruction after the branch will be fetched, in this case a NOP, and afterwoods instead it will fetch the SUBI instruction, this is because the BPU is present and will fetch the correct one without losing any other clock cycles, and also the last fetch will be wrong, infact the signal WRONG FETCH is high twice. in **Figure 3.6** we can see the waves in the forwarding unit, and has we espected the states change from a STALL, stopping the branch in DECODE, to a forward state, precisely to a MEM to ID forwarding, and passing afterwards to a NO FORWARDING state.

- For the last kind the BRANCH instruction needs the data that will be loaded from the memory from the previous instruction. So the case is very similar to the previous one, but in this case the branch will need to stay in the DECODE stage for three clock cycles instead of two.

INSTRUCTION	1	2	3	4	5	6	7	8
LW R1, 1(R0)	IF	ID	EXE	MEM	WB	-	-	-
BNEZ R1, LABEL	-	IF	S	S	ID	EXE	MEM	WB

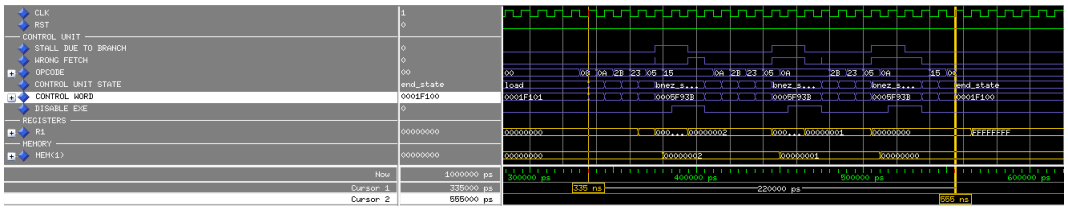


Figure 3.7: CU waves, stall and wrong fetch signals and relative result stored in the

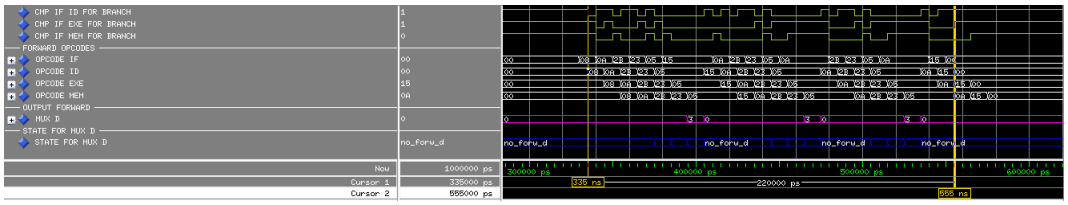


Figure 3.8: general waves, stall wave, stats assumed and relative output of the Forwarding unit

3.2 The structure of the Forwarding Unit

The Forwarding unit is composed of two main components: The FSM and the main structure that generates the correct signals for the FSM.
The data given as inputs are the Fetch and the Decode OPCODES, which are managed through internal pipelines to know at which stage they refer. Also other inputs are the addresses of the destination registers of the various stages in the Datapath, and also the required registers in the Decode stage. With this data is possible through equality comparators to understand the dependence.
So at this point the various outputs are provided to the Datapath and the Control Unit.

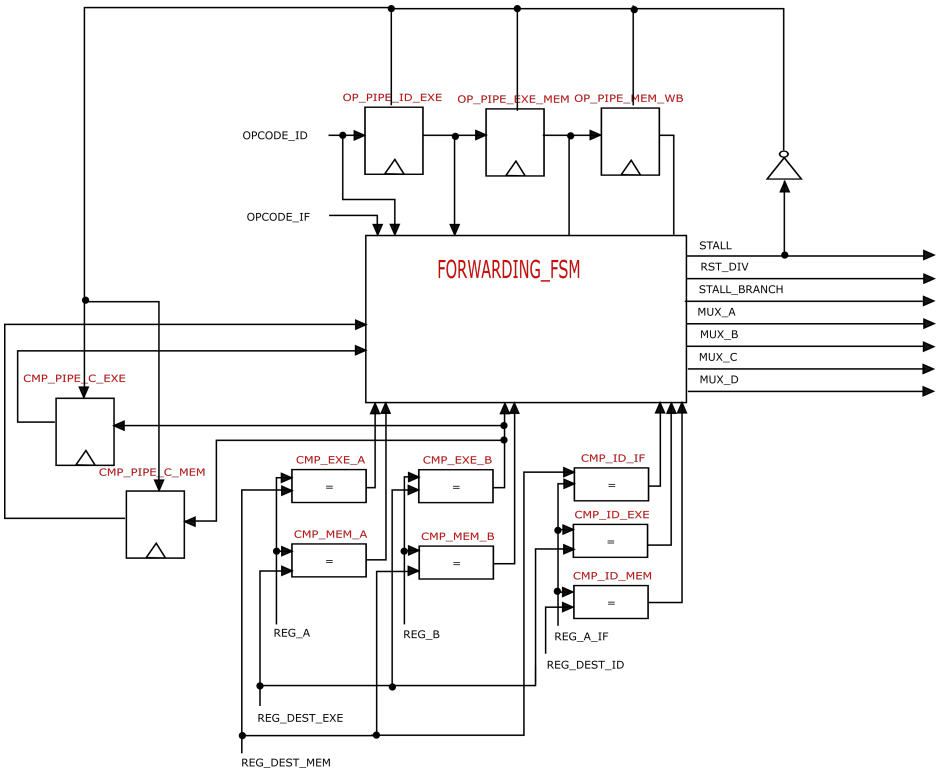


Figure 3.9: Structure of the Forward Unit that has an FSM to manage the forwarding

3.3 Control Hazards

Control Hazards have been also managed, these occur after a branch instruction, since only in the DECODE stage is possible to know the outcome of the result. So possible fetches can be wrong, but in this case the Control Unit will acknowledge this case, as it has been displayed before, and will annihilate in the DECODE stage the wrong fetch.

3.4 Structural Hazards

Structural Hazards have not been taken into account, since this is a LOAD and STORE architecture this kind of Hazard can occure only if the FETCH stage and the MEMORY stage try to access the memory at the same time. But since different kinds of memory have been used this is not possible, so this Hazard can never occur.

Chapter 4

The instruction cache

As it has been overviewed before in the DLX PRO a cache for the instructions has been added. This has been described in a complete structural way, implementing a direct mapped cache, where for each array in the memory eight instructions are stored and the total memory is composed of 16 array, so the cache can store maximum 128 instructions before needing to replace any old one, so the size of the cache is 512 kByte. In case the program in composed by more than 128 instructions some old ones will be overwritten, the ones that will be erased are the instructions that present the same lower seven bits of the new address to save.
The schematics of the component are represented by the image below:

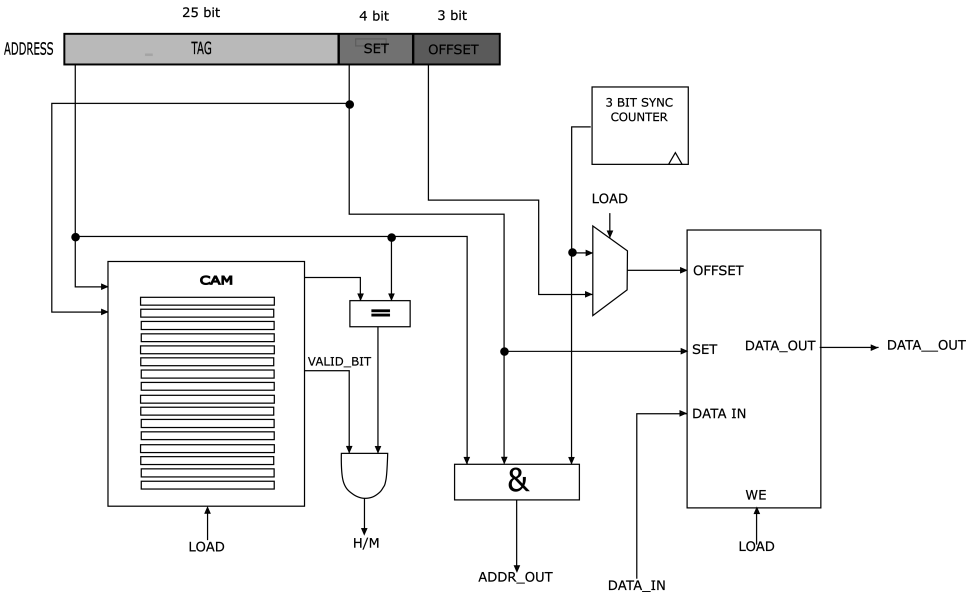


Figure 4.1: The instruction cache

A CAM memory is allocated that will receive as input the middle 4 bits generating on the output the relative TAG bits that have previously saved, if no data was previously saved then the valid bit for the relative TAG is '0' so even if for casualty the TAG requested corresponds due to the uninitialized value stored in the CAM there will be anyway a MISS event.
Regarding the result stored in the data cache, if a HIT event has occurred then the SET bits will select the corresponding line and the OFFSET bits will select the correct instruction between the eight present in the line. On the other side if a MISS event occurred then the Control Unit will acknowledge this and active the LOAD signal that will set high the write enable signals (WE) and will switch to the other input the multiplexer. The address to read in memory will be generated by the concatenation of the TAG, SET and the three bits of the counter, so that at each clock cycle the next adjacent data will be gathered. For this purpose we suppose that the instruction RAM is allocated for each address one byte, so 32 accesses to memory are required, and supposing that the memory has a throughput of one byte each clock cycle.

Chapter 5

The divisor

The 16-bit integer divisor has been designed through a non-restoring, 22 clock cycles, division algorithm. It has been implemented by an FSM control unit managing the structural datapath shown on **Figure 5.1**. The datapath is in charge of perform sign correction on both the inputs Z (the dividend) and D (the divisor), a series of sums and subtractions settings, and an eventual final correction on the sign of the output. This last one is provided through a 32-bit signal whose 16-bit upper part represent the final remainder, and whose 16-bit lower part represent the quotient Q. The final result is provided by approximating it to the nearest lower integer. In order to save area, sum, subtraction and complementation operations are performed by the P4 adder, whose inputs/output are connected to the outputs/input of the divisor in the ALU unit through multiplexers properly controlled on the right position.
While no division is decoded, the reset signal is driven high by the forwarding unit. In this state, that corresponds to the LOAD state, all registers are enabled. When a new division is decoded, the reset signal is driven low, and in the next clock cycles the 16-bit inputs are sampled.
The first two states are necessary to perform the 2's complement on the negative inputs. This is due to the fact that the divisor has been designed initially only for positive operands, and then, it has been upgraded in order to manage also negative values. The former state, D_SIGN_STATE, is in charge of: extracting the absolute value of D, to save the initial value of Z, and to save, through a XOR gate, which sign the final outputs will have. The absolute value of D is obtained by connecting it to the second input of the adder through ADD_IN_D_MUX, a 0 to the first input of the adder through ADD_IN_R_MUX, and the last bit of D to the SIGN input of the adder. If D is negative, it will be complemented, otherwise not. Z input and XOR_SIGN signal are saved in order to avoid a change of the inputs while the divisor is still running. Saving is obtained by setting to 0 the enable signal of the relative registers. The latter state, Z_SIGN_STATE, is in charge to save the absolute value of D, and to perform the same operation seen before, on the input Z, by properly setting the input muxes of the adder. The obtained result is sent to Q register in order to

initialize it to Z value, as required by the algorithm. The remainder signal R, as well, is initialized from the start of the computation to 0 value.
From the next clock cycle to the following 16 cycles, the non-restoring algorithm take place. The relative state is LOOP_STATE. During it, R and Q signals are concatenated into RQ signal and shifted left by one position to generate RQ_SHIFTED signal. The 16-bit of R are sent to the first input of the adder, D is sent to the second one, and a sum or subtraction between them will be performed dependently on the sign of the non-shifted R signal (R(15) = 0 for sum, R(15) = 1 for subtraction) , that is connected through a NOT gate to the sign input of the adder. Of course, the control unit properly set R_MUX_IN, Q_MUX_IN and the input muxes of the adder on the relative position. Then, the output of the adder is sent to the remainder, its last bit is negated and sent to the first bit of the quotient, whose upper bit are updated by the actual value of the lower part of RQ_SHIFTED. As will be explained better by the flowchart presented below, the algorithm iterates 15 times plus a last one, that corresponds to LAST_LOOP state, in which, at the end of the last addition, the sign of the next remainder , that corresponds to the actual output of the adder, is evaluated. If the remainder is negative, a restore will be performed in the next cycle during the RESULT_STATE_2. Otherwise, nothing has to be done, but one clock cycle will be wasted during RESULT_STATE_1.
The last two cycles corresponds to the ones necessary to restore the correct sign of the output, respectively Q_SIGN_STATE and R_SIGN_STATE. Having saved the sign difference generated by the XOR gate, now is possible to evaluate if the positive quotient Q and remainder R have to be negative. This is performed by connecting the output of the XOR register into the sign input of the adder, and proceed in a similar way seen for D and Z operands.
Finally, after 22 clock cycles, the correct quotient Q and remainder R are available at the output and a new division can take place.

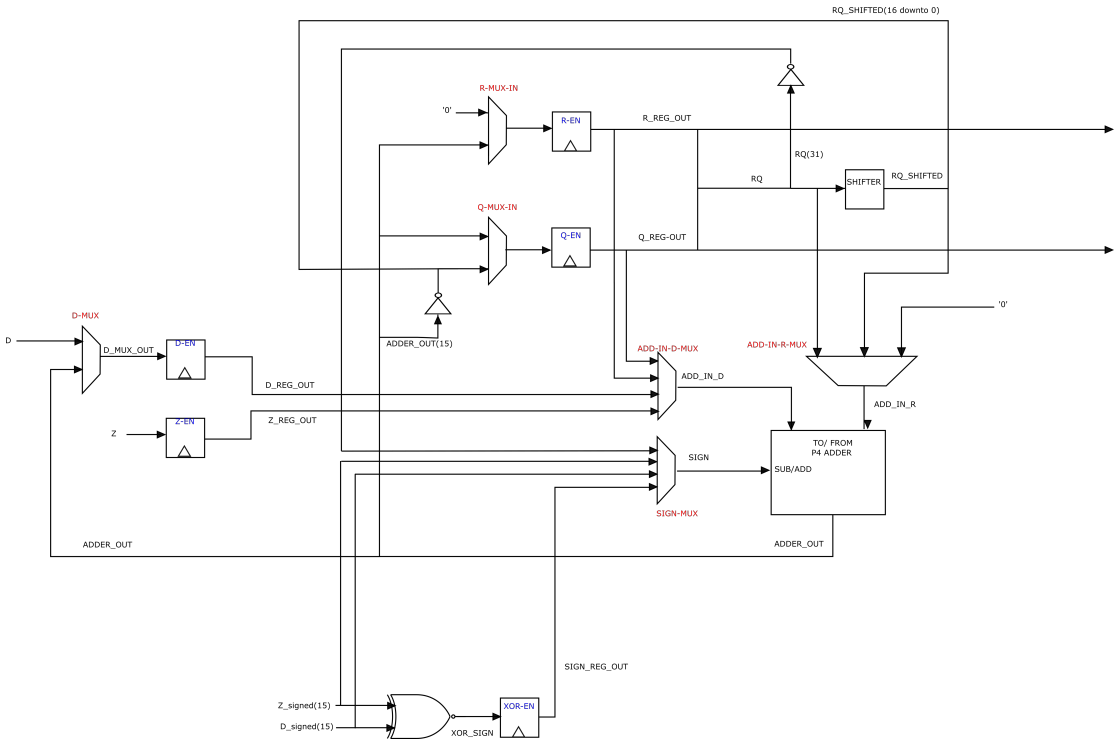


Figure 5.1: Datapath of the Divisor

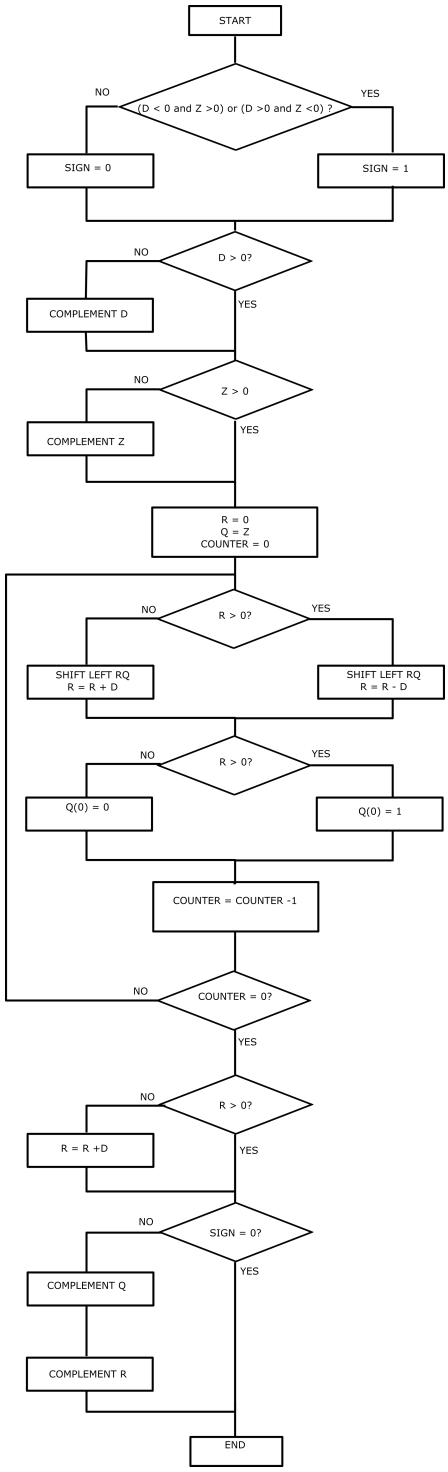


Figure 5.2: Flowchart of the Divisor

Chapter 6

Branch prediction unit (BPU)

Up to this point a static prediction was used for branching (the branch was always considered untaken). But this feature has been improved by using a dynamic branch prediction, so to have this feature another component has been implemented, the branch prediction unit (BPU). For this component a small FIFO cache has been realized, with four entries, and the prediction is saved into four different FSMs.
At the beginning there are no branches stored inside, so in case of a new one this will be predicted as before, that means not taken, but this will also be stored in an entry, so that when this branch will be present again the BPU, this will be able to predict the branch. When a new branch is stored, the counter increases so that it will point to the next entry for the next branch. Also the FSM relative to the entry is updated.
If the branch is already present in the BPU, then the component will drive the multiplexer, at the entry of the program counter, to the output of the BPU, and will set it to the stored value in the past.
But we must also keep into account that the BPU might predict the wrong value, and it's possible to know the exact result only when the branch is in the decode stage. If this event occurs, then the BPU will update the correct value of the FSM of the relative entry, and will inform the Control unit to flush the wrong instruction, so that the instructions will flow the correct way.
So now in cases of loops, only twice the prediction will be wrong, and in all the other cases no clock cycle will be lost, thus increasing considerably the performances of the processor.
The FSMs are all initialized to NOT TAKEN, and when the new entry is introduced in the BPU, the relative FSM passes to TAKEN or STRONGLY NOT TAKEN, depending on the result of the branch.
Also the BPU has its own Control Unit, that will compute if the evaluated branch needs to be stored, in case a miss occurred, in the cache, or if it needs to be updated, if was already present inside.

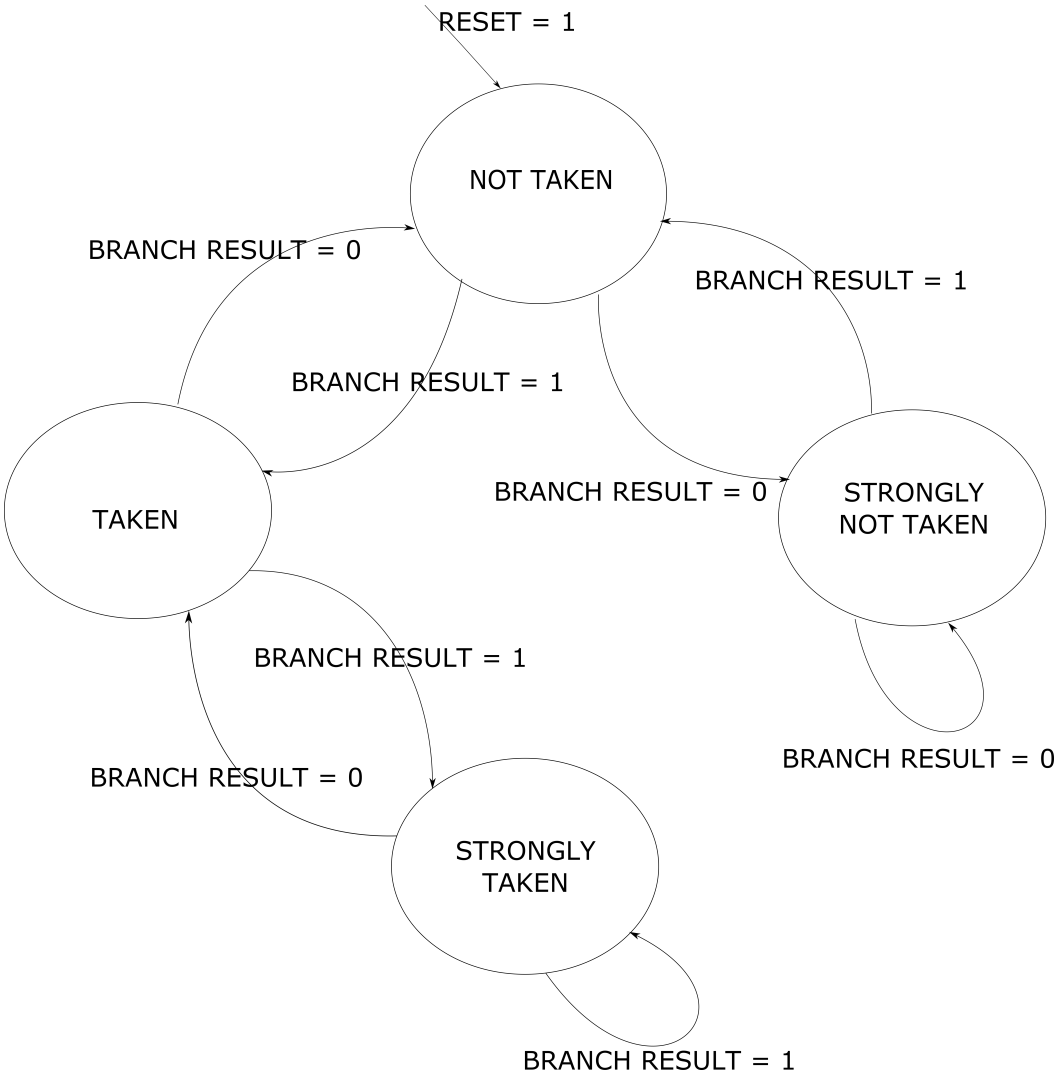


Figure 6.1: Structure of the Forward Unit that has an FSM to manage the forwarding

6.1 Datapath of the BPU

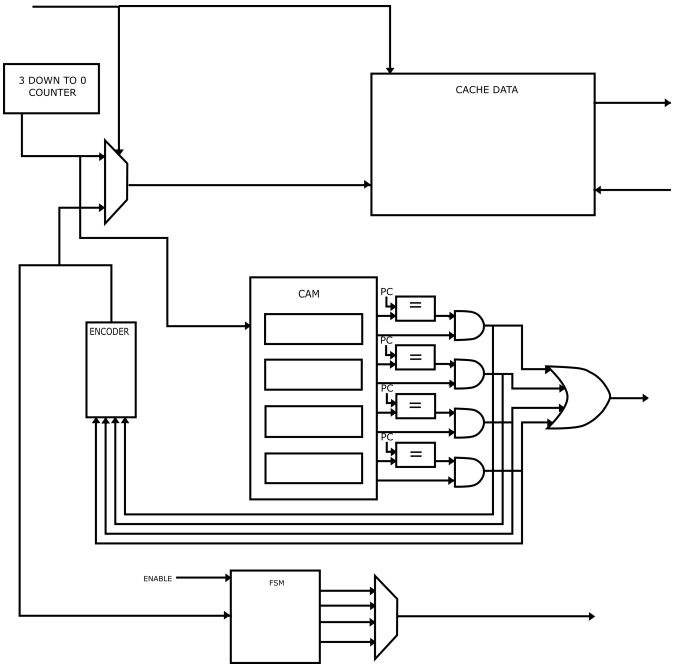


Figure 6.2: Datapath of the BPU

The CAM has on all the four outputs the values stored, these outputs are all simultaneously compared with the program counter and afterwards will be anded with the valid bit, since it's possible that a random value could generate a positive comparison, then all the values are inserted in an OR gate that will generate the HIT/MISS signal.
If a HIT occurs the multiplexer will switch to the output of the encoder, that uses the values of the AND gates to generate the correct address in the cache data, the WE signal will be low, and so the cache data will give as output the next program counter.
If a MISS occurs the multiplexer will switch to the output of the counter, that has increased its value by one, and the WE signal will be high, so that the new value will be stored in the cache data.
In the second part of the circuit the same encoder will select the correct FSM that will be updated, in case of a HIT event, or will be generated, in case of a MISS event, and afterwards the multiplexer will select the output of the selected FSM that will drive the multiplexer on the program counter input.

6.2 Control Unit of the BPU

The BPU has its own control unit to manage all the possible cases.
In case of a branch occures and the BPU didn't have the entry already stored, then the Control Unit will nedd to store the values inside, that means the destination of the branch and the program counter of the instruction. In case a branch occures, and the BPU found an entry inside, this means that a HIT occured, then the value of the FSM needs to be updated.

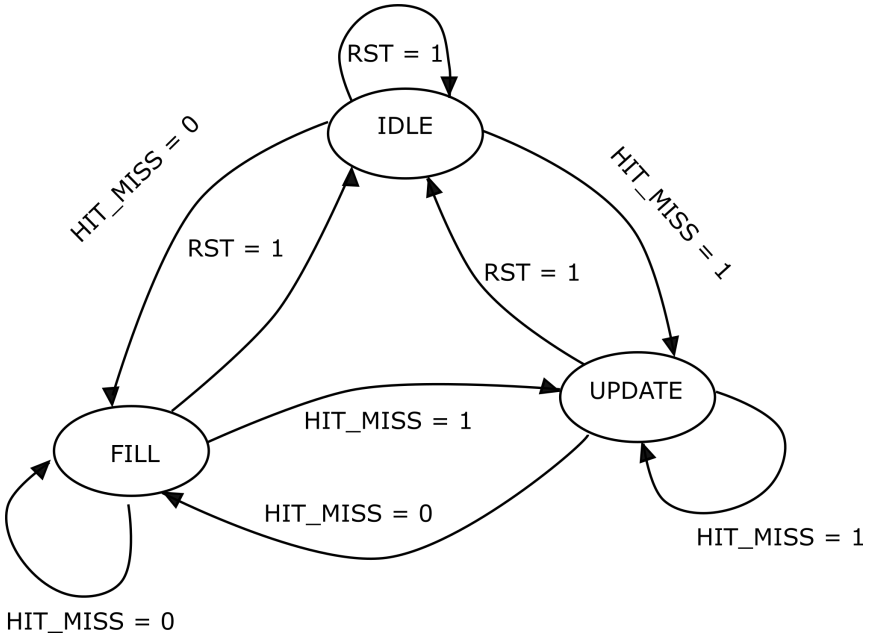


Figure 6.3: Control Unit of the BPU

Chapter 7

Power optimization

After implementing all the hardware to increase the performance, the power usage has been taken into account. The concept is to not generate switching activity in components that aren't been used, like various ALU components and the branch comparator.
A possible solution to this would be to perform a bitwise AND of the inputs of the component with an enable signa, annihilating so the values in input. But this will give as a result no switching activity only if in the previous moment the inputs would have been void.
So for this reason another approach has been used, instead of an AND a D latch has been used, which will have ad output the same input if the enable signal is high, and will keep the output constant in case of a low value of the enable signal. This approach is also known as Guarded Evaluation.

Chapter 8

Benchmark and simulation

To check the correct behavior of the circuit a Benchmark has been generated. For this purpose a digital filter algorithm has been chosen.
To create the benchmark the values have been initialized in the memory, this is performed with a loop while the reset is high, and also the results of the test are stored in the memory. The algorithm can be explained with the following formula:

$$Y[n] = X[i] * a + X[i-1] * b + X[i-2] * c \tag{8.1}$$

The value of i is initialized to 2 and the number of samples are 12, so it will produce 10 results, the values of the coefficients are:

$$a = 4, b = 0.5, c = 5 \tag{8.2}$$

Since the factors of **a** and **b** are a power of 2 some simple shifts are used, while for multiplying for the coefficient **c** a mult instruction has been used.
The code used is here described:

```
add r1, r0, r0          ;inizialize x
addi r2, r0, #10        ;loop counter
addi r31, r0, #5        ;constant value for c
add r30, r0, r0         ;set value to start saving in
    memory
label_1:
lw r3, 0(r1)
slli r4, r3, #2         ;multiply by 4
lw r5, 1(r1)            ;load second value
srai r6, r5, #1         ;divide by 2
lw r7, 2(r1)            ;load third value
mult r8, r7, r31        ;multiply by c
add r8, r6, r8          ;first addition
add r8, r4, r8          ;second addition
sw 12(r30), r8          ;store the data
addi r30, r30, #1       ;increase the address for storing
```

```
addi r1, r1, #1          ;increase index for the next
    sample
subi r2, r2, #1          ;decrease the loop counter
bnez r2, label_1
nop
```

This benchmark has been chosen for two main reasons: for showing the correct behavior of some PRO instructions, and to show the correct behavior of the BPU.
In the Appendix it's possible to see all the images of the waves of this simulation. First thing that it's possible to notice in **Figure 11.1** is the behavior of the cache, how it's possible to see at the beginning there will be a miss event, since the cache is empty, but after the loop there will be only hit events, so the instructions will be performed one after the other one without interruptions.
After the inizialization of the variables, at the beginning of the loop, pratically after the label *label_1*, we can see the dependecy between the load instruction and the shift, this will need a data forwarding after a stall. This phenomenon is displayed in **Figure 11.3**, **Figure 11.4** and **Figure 11.5**, where the branch state remains for two clock cycles in the DECODE stage.
From this benchmark it's also possible to check the correct behavior of the BPU, where in the first loop the prediction will be mispredicted, since the entry is empty, and will also be mispredicted at the last loop, since it will be in a strongly taken condition but the branch will result not taken.
It's also possible then to compute the misprediction ratio of this simulation, from the formula:

$$misprediction\,ratio = \frac{\#\,mispredicted\,branches}{total\,\#\,of\,branches} \tag{8.3}$$

So in this case, since the numbers of loop are 10, so the branches are also 10, we have a misprediction ratio of 20%, while in the basic version, with a static prediction, we would have got a misprediction ratio of 90%, thus resulting in not losing 7 clock cycles.
Some others computations have been done to check the improvement in the performace of the DLX, in **Figure 11.6** it's reported the number of clock that the BASIC DLX would have required, for a total of 338 clock cycles counting the inizialization. While for the PRO version, tabel in **Figure 11.7**, it would have required 238, thus meaning an improvement of about 30%.
With the final version having a BPU another 7 clock cycles are not lost, so it requires 231 clock cycles, thus meaning another 2% improvement, with a total improvement in performace of about 32%.
The BPU advantages are more significant for nested loops and with higher counters, while here the result of increasing performace is not very marked.

Chapter 9

Synthesis

After testing the correct behavior from the logical point of view, the design has been synthesized using the Synopsys tool Design Vision. For this step the momory has been removed so that we could have a fully synthesizable DLX.
As first step it has been decided to remove also the cache, since it should not introduce a critical path and it would let the synthesis be much faster. The data saved from the design are reported in a tabular form below:

MODIFICATION	ARRIVAL TIME	DYNAMIC POWER	LEAKAGE POWER	AREA	CONSTRAIN
NO MODIFICATION	2.75 ns	3.9254 mW	442.6020 uW	22078.265625	NO
WITH CSA	2.65 ns	3.9254 mW	448.2681 uW	22367.408203	NO
CONSTRAIN 1	2.56 ns	4.5293 mW	448.1323 uW	22369.269531	2.6 ns
CONSTRAIN 2	2.26 ns	5.1229 mW	458.8427 uW	22580.207031	2.3 ns
CONSTRAIN 3	1.96 ns	5.8985 mW	486.1872 uW	23165.673828	2 ns
CONSTRAIN 4	1.76 ns	6.5668 mW	512.1296 uW	24030.173828	1.8 ns
CONSTRAIN 5	1.66 ns	6.9629 mW	521.9282 uW	24279.681641	1.7 ns
CONSTRAIN 6	1.65 ns	7.4024 mW	531.7540 uW	24565.898438	1.6 ns

So it's not possible to have a critical path less than 1.66 ns so a maximum frequency of 600 MHz. So keeping into account power we set a maximum power of 6 mW with a warking frequency of 500 MHz.
But lets summarize better the step done during the synthesis procedure:
At each step a new bound has been set for the critical path, asking the synthesizer to respect this kind of constrain, producing though a which power consumption and a higher area, all the percentages are computed with respect to the basic synthesis of the chip.

- The first synthesis was just to understand where the critical path is, and from the data received it has been found to be in the DECODE stage, passing through the address adder for the program counter. So it has been decided to substitute this adder with a faster one, so a carry select addder (CSA) has been implemented.
  With this variation it has been possible to achieved a lower delay but a slightly higher leakage power and area, to be precise:

  - Critical path passing through a pipeline stage of the multiplier
  - Delay reduced by 3.77%
  - Dynamic power increased by 0%
  - Leakage power increased by 1.28%
  - Area increased by 1.31%

- Afterwards some constraines have been imposed, the first with a 2.6 ns constrain gave the following results:

  - Delay reduced by 7.42%
  - Dynamic power increased by 15.38%
  - Leakage power increased by 1.25%
  - Area increased by 1.32%

  The slack is met

- Another constrained set this time to 2.3 ns gave the following results:

  - Delay reduced by 21.68%
  - Dynamic power increased by 30.51%
  - Leakage power increased by 3.67%
  - Area increased by 2.27%

  The slack is met.

- Another constrained set this time to 2 ns gave the following results:

  - Delay reduced by 40.31%
  - Dynamic power increased by 50.26%
  - Leakage power increased by 9.85%
  - Area increased by 4.92%

  The slack is met.

- Another constrained set this time to 1.8 ns gave the following results:

  - Delay reduced by 56.25%
  - Dynamic power increased by 67.29%
  - Leakage power increased by 15.71%
  - Area increased by 8.84%

  The slack is met.

- Another constrained set this time to 1.7 ns gave the following results:

  - Delay reduced by 65.66%
  - Dynamic power increased by 77.38%
  - Leakage power increased by 17.92%
  - Area increased by 9.97%

  The slack is met.

- Another constrain setted this time to 1.6 ns didn't meet the slack to it's not possible to decrease the delay more than 1.7 ns.

So after analyzing this data we decided to keep into account power consumption choosingthe third constrain having a working frequency of 500 MHz (40.31% faster), a dynamic power consumption of 5.8985 mW (50.26% higher), a Leakage power consumption of 486,1872 uW (9.85% higher) and a total area of about 23165 (4.92% bigger)

Chapter 10

Physical layout

PHYSICAL DESIGN By employing the CAD tool Encounter by Cadence it has been possible to proceed with the generation of the layout of the DLX. After generating, with the synthesis tool, the necessary files, such as the .sdc files containing the necessary info for clock and combinational delay constraints, and the Verilog netlist.
The layout has been created by starting from a .globals file. It indicates to Encounter the location of the verliog post synthesis netlist (DLX.v) , the reference to the layout of the physical cells (.lef file), the definition of power supply nets (**VDD** and **GND**) and a reference to a file for MMMC analysis (default.view). The various phases of the design of the layout can be grouped as: Floorplan structuring: It has been defined the area of the CORE and the area for the power ring channel by selecting Core to die boundary and forcing 4 micron from the four side of the core. Power ring and power stripes insertion: Two metal rings (one for vdd one for gnd) has been inserted into the channel defined previously. Metal 10 and metal 9 have been assigned respectively for vertical and horizontal lines. Then, 13 vertical power stripes have been inserted on metal 10, with a set to set distance of 20 and a relative from core distance of 20 for geometrical reason. Standard cell power routing: Horizontal wires for cells power supply connection have been inserted and connected to the power ring and to the vertical stripes. Placement: To each cell is assigned a position. Pre clock tree synthesis optimization: Optimize the design just before the clock synthesis Clock tree synthesis: Synthetize the clock tree after generating automatically the .ctstch file. Post clock tree synthesis optimization: Optimize the design just before the routing. Place filler: These step fills the holes without cells with N+ and P+ wells. Routing: The connection between the cells is made. This is performed by employing first a trial route in order to plan the wire position, then the true route is made Post routing optimization: Optimize the design once the routing is done. The resulting layout is shown In **Figure 10.1**. The figure after shows the area occupied by the cache on th chip. The results obtained after the physical design, in terms of parasitic resistance and capacitance have been reported into .spf ,.spef, .setres and .setload files. In terms of area and gate

count, the results have been reported into .gatecount and .sdf files. Finally, an histogram shown in **Figure 10.3**, reporting timing slacks, has been generated, showing that no violation is present in the design.

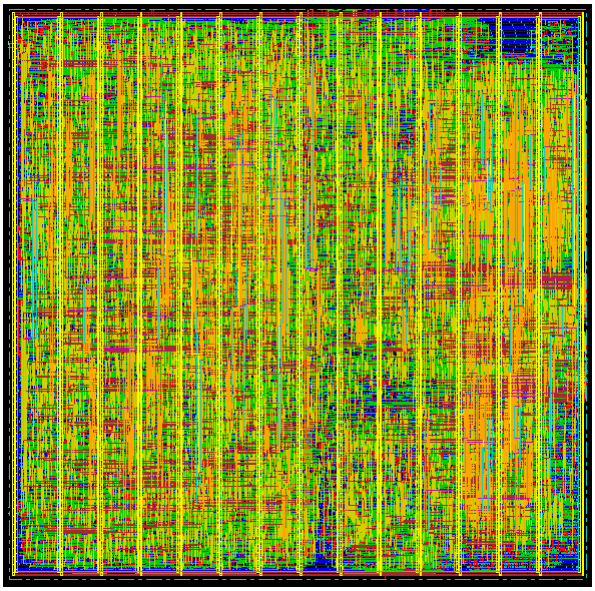


Figure 10.1: Physical layout of the chip

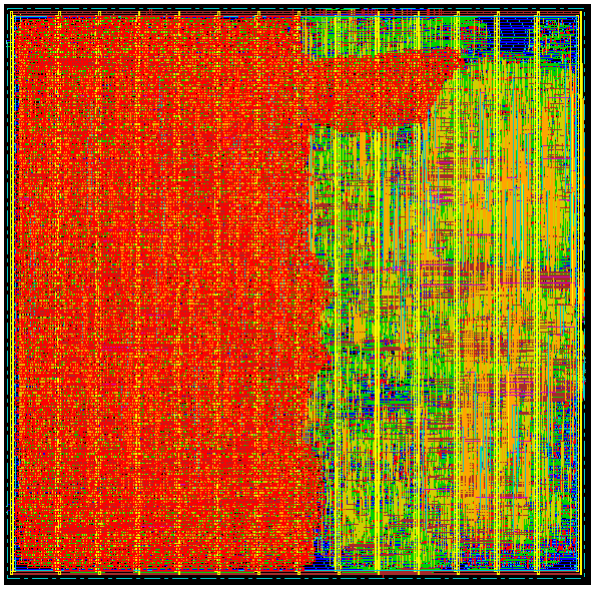


Figure 10.2: Physical layout of the chip showing the cache

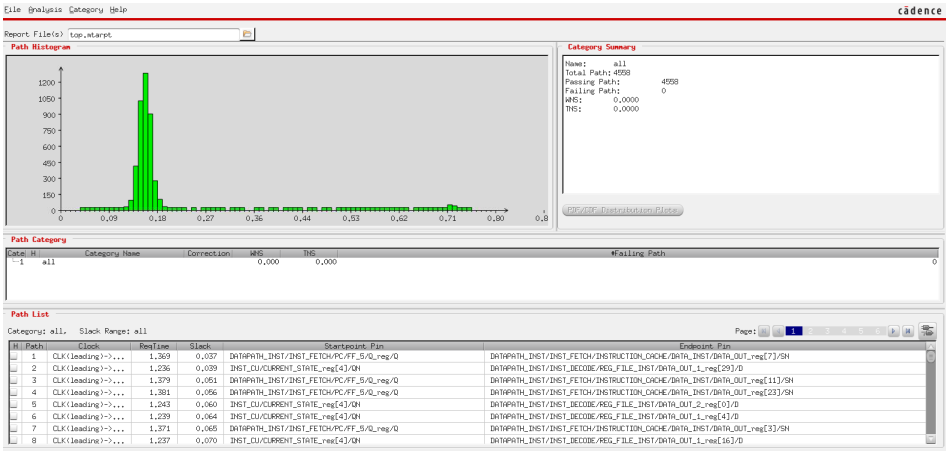


Figure 10.3: Slack distribution

Chapter 11

Appendix

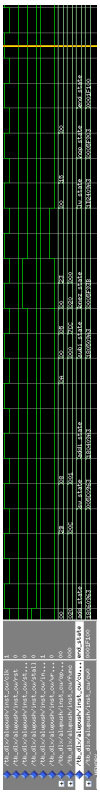


Figure 11.5: Final misprediction

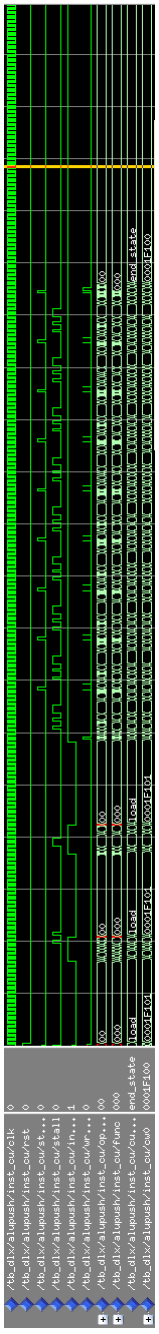


Figure 11.1: General overview of the waves

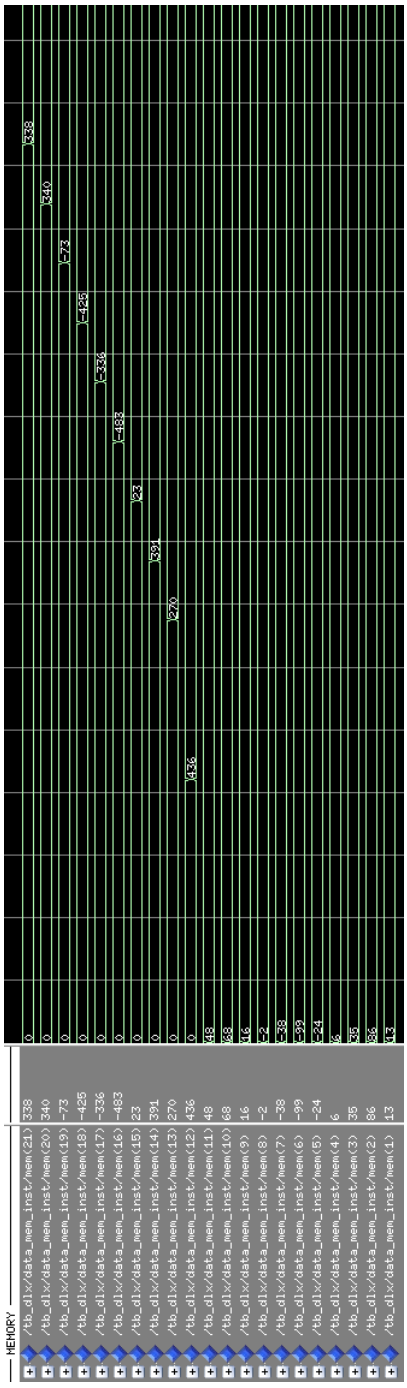


Figure 11.2: Results stored in memory

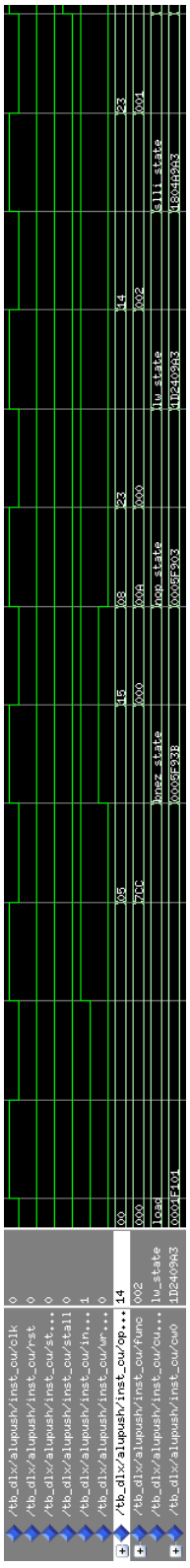


Figure 11.3: The first branch is predicted wrong

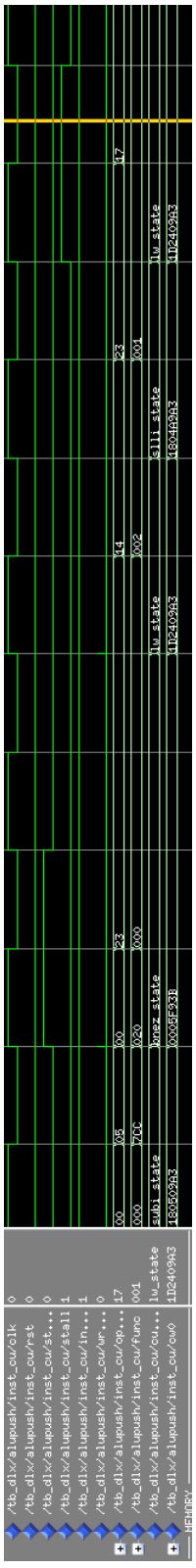
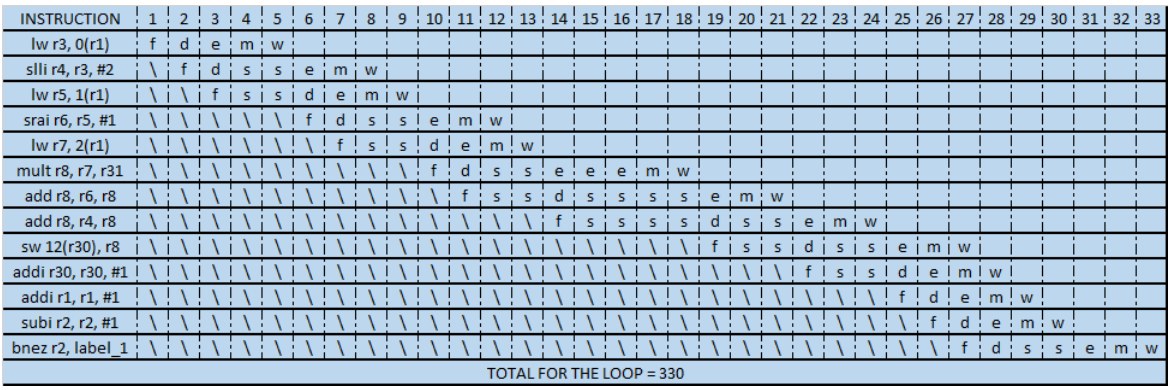


Figure 11.4: Right prediction of the branch during the loop



INSTRUCTION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
lw r3, 0(r1)	f	d	e	m	w																												
slli r4, r3, #2	\	f	d	s	s	e	m	w																									
lw r5, 1(r1)	\	\	f	s	s	d	e	m	w																								
srai r6, r5, #1	\	\	\	\	\	f	d	s	s	e	m	w																					
lw r7, 2(r1)	\	\	\	\	\	\	f	s	s	d	e	m	w																				
mult r8, r7, r31	\	\	\	\	\	\	\	\	\	f	d	s	s	e	e	e	m	w															
add r8, r6, r8	\	\	\	\	\	\	\	\	\	\	f	s	s	d	s	s	s	s	e	m	w												
add r8, r4, r8	\	\	\	\	\	\	\	\	\	\	\	\	\	f	s	s	s	s	d	s	s	e	m	w									
sw 12(r30), r8	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	s	s	d	s	s	e	m	w					
addi r30, r30, #1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	s	s	d	e	m	w					
addi r1, r1, #1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w				
subi r2, r2, #1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w			
bnez r2, label_1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	s	s	e	m	w
TOTAL FOR THE LOOP = 330																																	

Figure 11.6: Flow of the instructions for the basic version

INSTRUCTION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
lw r3, 0(r1)	f	d	e	m	w																		
slli r4, r3, #2	\	f	d	s	e	m	w																
lw r5, 1(r1)	\	\	f	s	d	e	m	w															
srai r6, r5, #1	\	\	\	\	f	d	s	e	m	w													
lw r7, 2(r1)	\	\	\	\	\	f	s	d	e	m	w												
mult r8, r7, r31	\	\	\	\	\	\	\	f	d	s	e	e	e	m	w								
add r8, r6, r8	\	\	\	\	\	\	\	\	f	s	d	s	s	e	m	w							
add r8, r4, r8	\	\	\	\	\	\	\	\	\	\	f	s	s	d	e	m	w						
sw 12(r30), r8	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w				
addi r30, r30, #1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w				
addi r1, r1, #1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w			
subi r2, r2, #1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w		
bnez r2, label_1	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w	
nop	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	\	f	d	e	m	w
TOTAL FOR THE LOOP = 230																							

Figure 11.7: Flow of the instructions for the PRO version

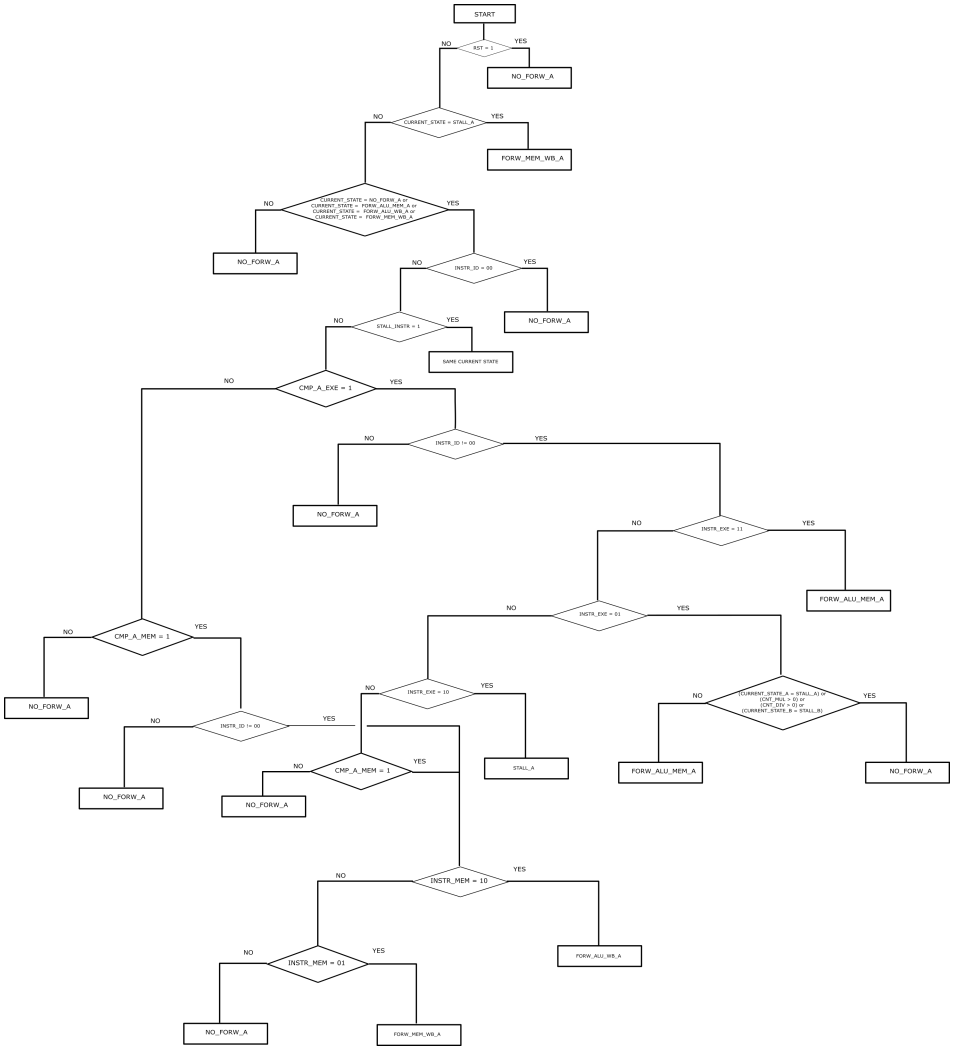


Figure 11.8: Flowchart for MUX A

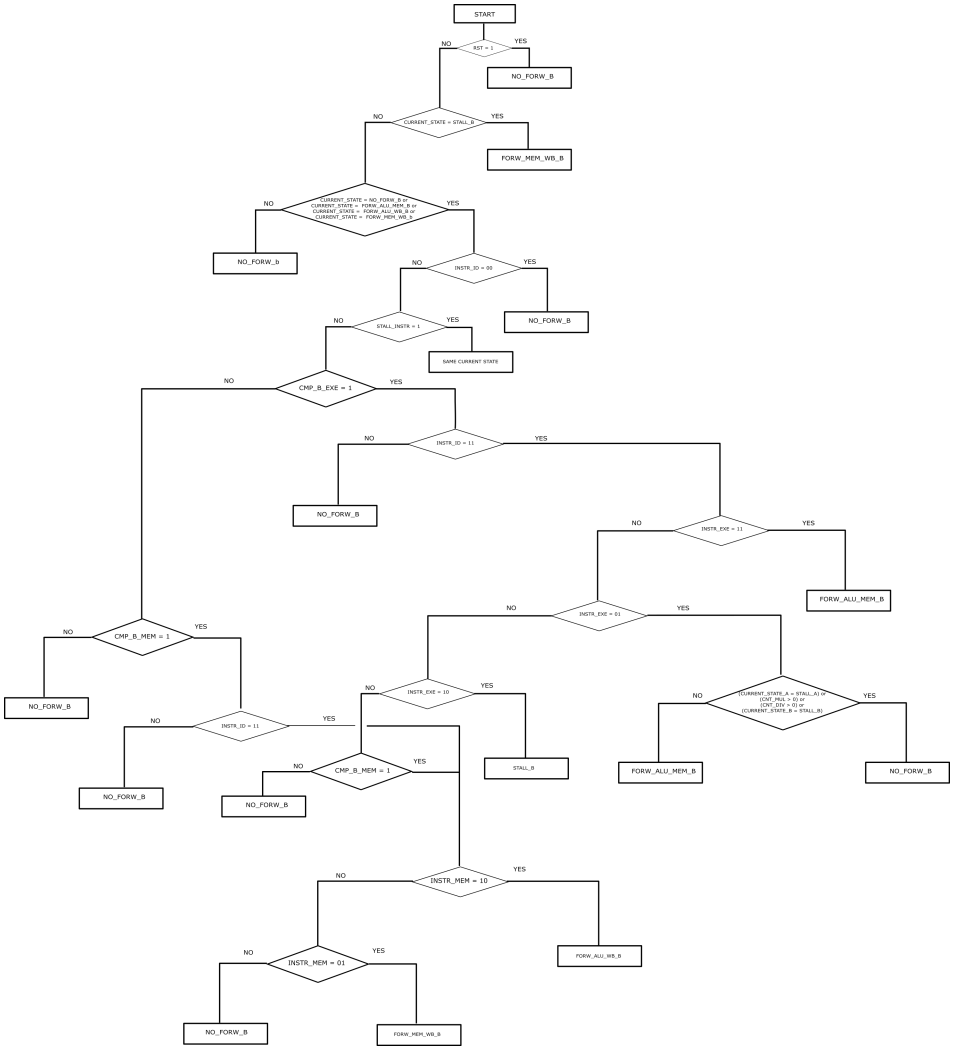


Figure 11.9: Flowchart for MUX B

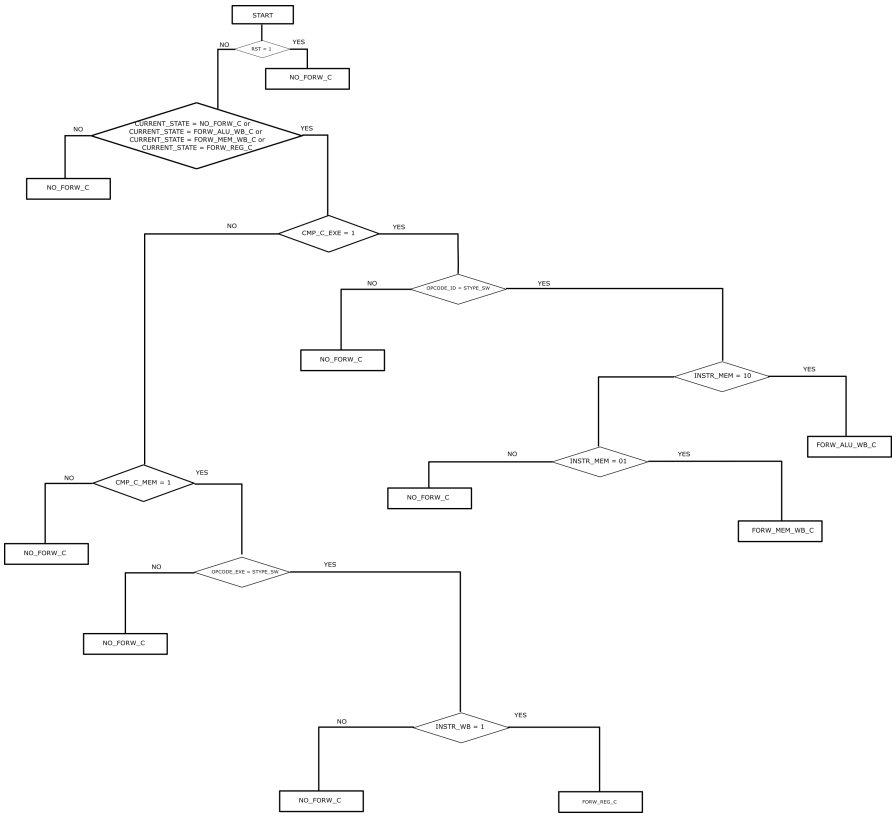


Figure 11.10: Flowchart for MUX C

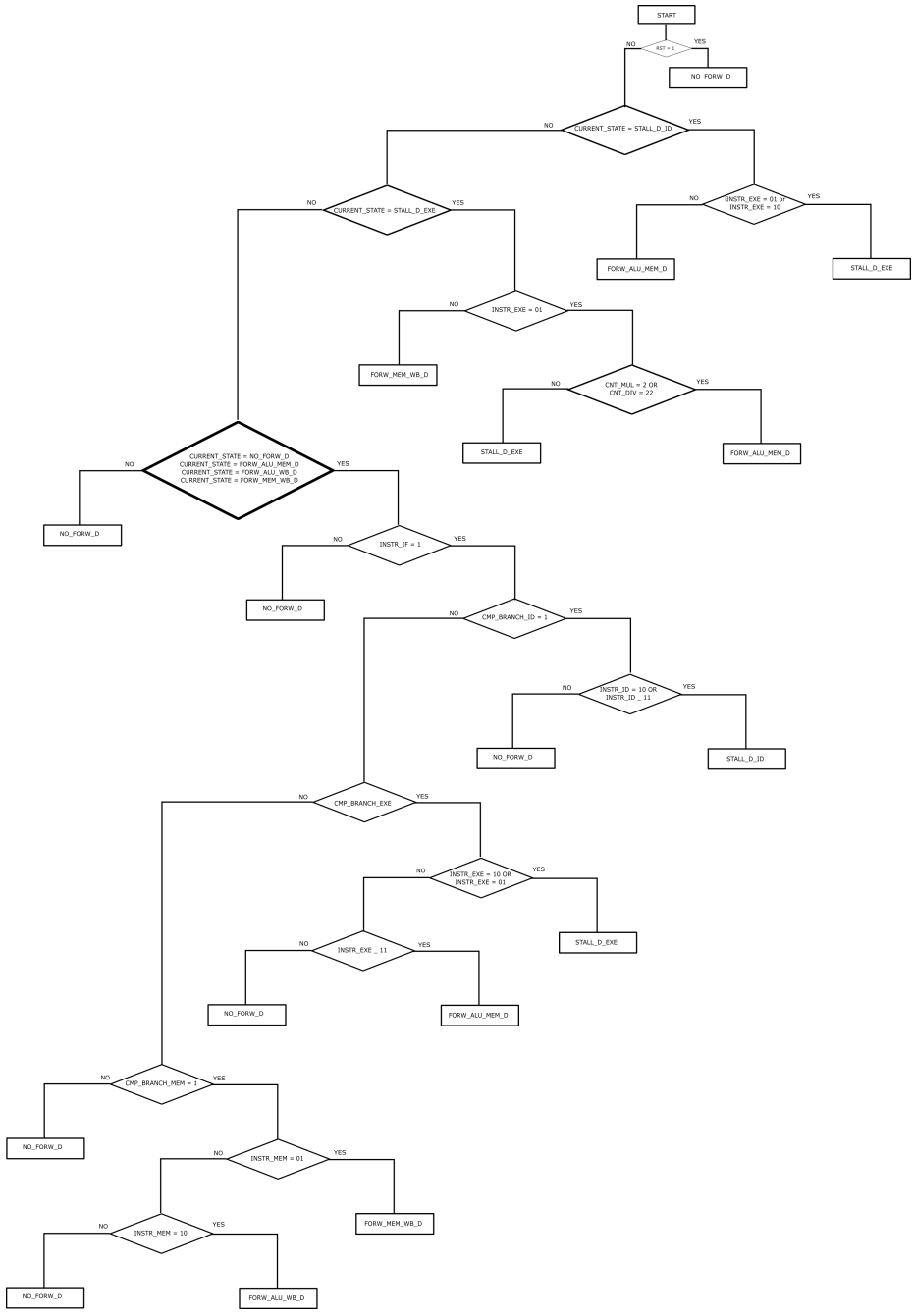


Figure 11.11: Flowchart for MUX D