Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group_51

Marco Chiarle, Giovanni Santangelo, Alessandro Vargiu

October 19, 2023

Contents

CHAPTER 1

Abstract

The DLX is a RISC architecture, a simplified with a mixture of capabilities from a set of RISC processors such as the Stanford MIPS CPU. It is a 32-bit load/store architecture. The goal of this project is to build a DLX implementation from scratch, using VHDL language. There are three main blocks that compose this architecture.

First (not in order of relevance), the Control Unit. The basis for our model had the Hardwired Control Unit. It also manages the pipeline in case of hazards. The Datapath is another important block of the DLX. Some internal blocks, like the Pentium 4 Adder were previously created during the course laboratories and were used for this final project. Another component is the T2 Shifter, which was shown in the course lectures and we decided to implement it. The Hazard Unit is another component that works alongside the Control Unit, to detect the different types of hazards that could occur during the execution of instructions in the pipeline. Later in this paper, all of its features are explained. After finishing the design phase, all the components were tested exhaustively using a bottom-up approach. Each component was singularly tested to verify its correct behavior, using VHDL testbenches. In the next testing phase, the entire CPU was tested using assembly programs and Modelsim environment to verify the waveforms. The next step was the synthesis phase, using Synopsys, which forms a Logic Synthesis of the CPU along with some optimizations. Thanks to this step, the step the maximum frequency of the implemented processor is 667 MegaHertz. The last step was the physical design phase, using Innovus. All the process steps will be described in this report, along with all the features provided by this DLX implementation.

CHAPTER 2

Introduction

In this short section the main concepts of the architecture are explained along with the related components. Additionaly, the added features which make up the Pro version of the DLX will be introduced along with related conceptual diagrams and schematics. Lastly, the chapter will conclude with a mention on how the rest of the documentation is organized.

2.1 DLX Overview

The DLX we implemented consists in the computation and propagation of 32 bit instructions. At a given time instant, multiple instructions of a partyicular assembly program are processed as this processor is constructed with a pipeline methodology. In particular, the stages of the pipeline for each instruction are given below:

1. Instruction Fetch;
2. Instruction Decode;
3. Execution;
4. Memory;
5. Write Back;

Furthermore, there are three fundamental block of the implemented DLX which are involved during the pipelined execution of the program:

1. Datapath Unit;
2. Control Unit;
3. Hazard Unit;

These components are explained in the following subsections of this chapter.

2.1.1 The Datapath Unit

The inclusion of the pipeline serves to increase the throughput of the overall program serving to reduce idle times of the processor. The pipeline mechanism is noted most clearly in the datapath and control unit which are the fundamental hierierchical components of the processor. In Figure 2.1 the datapath with a pipelined mechanism is depicted. The figure also shows also some external input signals which

are entering the datapath. These are control signals which come from another component called the Control Unit which sends appropriate control bits to the datapath depending on the instruction to be executed. Another important note about Figure 2.1 is that it depicts 2 memories, the instruction memory (IRAM) and the data memory (DRAM). These two memories are external components to the datapath and only interface it. Respectively, they serve an important role in the instruction fetch phase and memory phase of the instructions described later on.

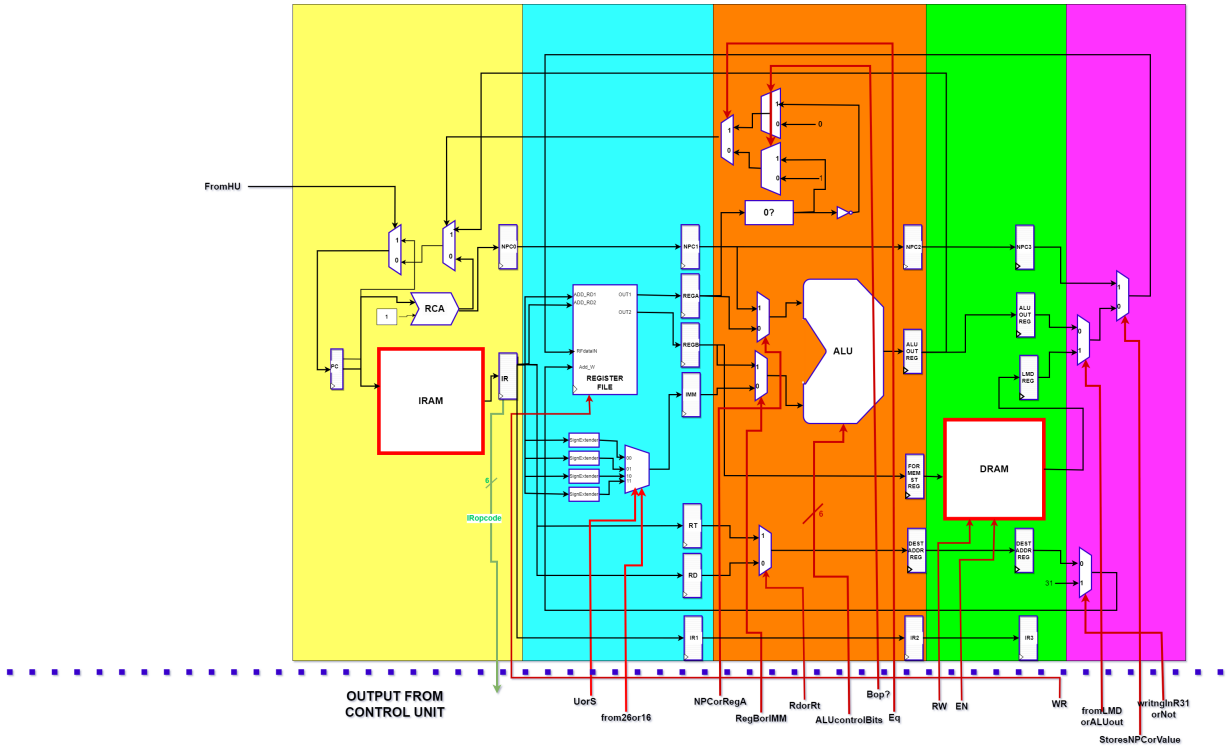


Figure 2.1: DLX datapath component

2.1.2 The Control Unit and Hazard Unit

The Control Unit and Hazard Unit are side components which serve to control. Moreover, the control unit is constructed to give orders to the datapath about which instructions need to be executed but it gives such orders alertive to possible hazards which could be detected by the Hazard Unit. They will be discussed in a following chapter of the document.

2.2 Pro Version Features

The presented DLX has the following capabilities:

1. Extended Instruction set Architecture;
2. Advanced Arithmetic Logic Unit;
3. Extra component, the hazard unit;

2.2.1 Extended Instruction set Architecture

The processor is able to run 50 instructions all listed in this section:

1. **RTYPE instructions**: ADD, SUB, AND, OR, SGE, SLE, SEQ, SNE, SRL, SRA, SLL, XOR, SLT, SGT, XNOR, NAND, NOR, ADDU, SUBU, SGEU, SGTU
2. **ITYPE instructions**: NOP, ADDI, SUBI, ANDI, ORI, BEQZ, BNEZ, LDW, STW, XORI, SGEI, SLEI, SLLI, SNEI, SRLI, SEQI, SRAI, SLTI, SGTI, ADDUI, SUBUI, XNORI, NORI, NANDI, SGEUI, SGTUI, JR
3. **JTYPE instructions**: JMP, JAL

The datapath, control unit and hazard unit are able to process additional instructions with respect to the basic set of instructions. some of the additional instructions involve the ability to distinguish signed operands with unsigned for some operations, extra logical functionalities, more compare functionalities, and the ability to jump to an address stored in a particular register.

2.2.2 Advanced Arithmetic Logic Unit

The inner components of the arithmetic Logic Unitc Unit are of advanced type. it is composed of a pentium 4 Carry LookAhead Spare Tree for more efficient addition and subtraction operations. In addition, the logicals have been designed d with the T2 Logic methodology. The relevant schematics of the introduced inner components of the arithmetic logic unit are analyzed in Execution Unit section of the report

2.2.3 Extra component, the Hazard Unit

This additional component is inside the DLX and it is able to make the distinguishment of the following type of hazards:

1. Read After Write Hazards
2. Control hazard for branch instructions
3. Jump detection

These hazards will be detected and given to the control unit which will actuate the stall for the datapath. More information on the hazard unit will be given later on in the report.

2.3 Structure of the report

The structure of this documentation consists in the thorough analysis of the components in the datapath based on the pipeline stages. Consequently an analysis on the implemented Control Unit and Hazard Unit is exposed explaining also their exhange of informatio. Lastly, an insight on the synthesis and physical design process is made available. In the next section we analyze the datapath starting from the components dedicated to the Fetch Stage.

CHAPTER 3

Fetch Stage

The fetch part of the datapath serves to do the operation of obtaining the current instruction. from the external Instruction Memory (where the program is stored) to then feed it to the subsequent unit of the datapath which decodes the bits. In this small section there will be the analyzation of the components of the datapath in this respective unit along also the external Memory from which the Instruction Register acquires the data.

3.1 Datapath Related Components

3.1.1 Program Counter Register

First of all there is the Program Counter Register whose purpose is to store the current address of the instruction inside the Instruction Memory from which the instruction is fetched. Additionally, this register is synchronous reset and enable to the clock. At reset time, the PC register stores the all zero value, meaning that there will always be the fetching of the instruction at the adress 0x00000000 at start up. Another important remark is about the signal of input of the PC depends on the outcome of two combinational multiplexers. One multiplexer decides to feed through the address of the next instruction or the address coming from the output of the ALU in the execute stage. This decision is made on the basis of a branchStatus signal determined during the Execution Stage, which takes value 1 when a jump or branch needs to be performed. As a second criteria, we included the second multiplexer to decide if it is needed to stall the PC register or not. This action depends on whether the hazard unit external to the datapath detects a hazard in the program. In such case it sends a selection signal for this multiplexer set to 1, for making the Program counter register assume the same address value in the subsequent clock cycle, so that a stall in the pipelined is formed halting the program.

Additionally, in order to compute the address of the subsequent instruction of the program, a ripple carry adder is used with a fixed operand of one added to the next instruction.

3.1.2 Instruction Register

This is the second important register which will store the value of the fetched instruction from the Instruction Memory. like the program counter register, it has a synchronous enable and a synchronous reset sensitive to rising edges of the clock. The output of this register is propagated and used for the nex important stage, the decode stage which is explained in the following chapter.

3.1.3 Next Program Counter

This important register is placed for the propagation of the subsequent instruction address for the following pipe stages. This way, if the instruction that was fetched is a jump or a branch, it will be able to compute the address of the jump later on in the execution stage.

3.2 External Instruction Memory

The instruction memory is the memory where all the instructions of the program are stored and it is the component from where the Instruction Register fetches the instruction. Upon the presence of an active high reset signal, it loads in a raw format all the thirtytwo bit instructions composing the program. While the reset is zero, on the basis of the adress that is felt at its input it outputs the associated instruction for the Instruction Register.

In Figure 3.1 is a conceptual drawing which is made to visualize the structure of this component more clearly. the main criterias to consider are that this memory will always store the first instruction of the program at the all zero address. The second feature of the memory is that it stores the whole instructions in a row of the memory, this leads to the reason why the addresses are incrememnted by one instead of four, and also it is the reason why the ripple carry adder adds one instead of four for computing the subsequent instruction's address. We designers made this decision for the purpose of easily creating the raw memory files while testing the datapath in the early stages. Also another small advantage of this choice is that the minimum required depth of the memory in thi way is exactly the amount of instructions of the assembly program

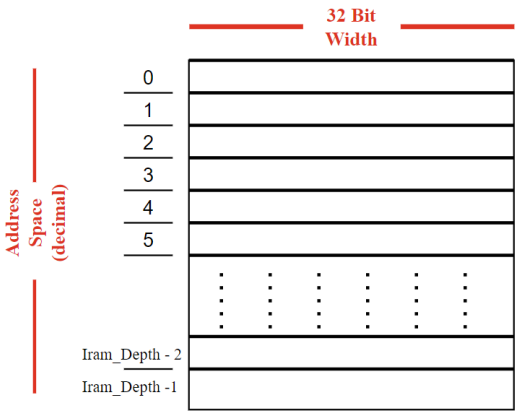


Figure 3.1: Instruction Memory with addresses separated by one as offset

CHAPTER 4

Decode Stage

In this part of the datapath the instruction that was previously fetched is broken down and given to several components. This part of the datapath contains more circuitry with respect to the fetch unit:

1. Sign Extender Module and 4to1 multiplexer;
2. Register File;
3. Pipe Registers for subsequent execution phase;

4.1 Sign Extender Module and four to one multiplexer

The sign extender module is designed with the final goal of interpreting correctly the immidate bit field stored in an immidiate type or jump type instructions. In fact, as the implemented DLX gives the possibility to distingush operations for unsigned and signed operations, this combinational block gives the possibility to extend the most significant bit of the immediate field of the immediate type instruction (sixteenth least significant bit) to the the other sixteen bits. The same is done for the immediate field of the Jump-Type instructions (this time containing a twentysix bit width instead of sixteen). The reason for this choice is because the DLX wires all have thirtytwo bit width, hence we must extend the original immidiate value of the instruction accodingly. Furthermore, there are four possible possibilities of interest that could be presented:

1. an Immidiate-Type instruction interpreting the immediate field as signed: in this case depending on the Most significant bit of the immediate bit field, the sixteen additional most significant bits will be either zero, or one. in this scenario we are extending a sixteen bit immediate to a thritytwo bit value.
2. an Immidiate-Type instruction interpreting the immediate field as unsigned: in this case regardless of the Most significant bit of the immediate bit field, the sixteen additional most signinfiant bits will be zero. In this scenario we are extending a sixteen bit immediate to a thritytwo bit value.
3. a Jump-type instruction interpreting the immediate field as as signed: in this case depending on the Most significant bit of the immediate bit field, the six additional most signinfiant bits will be either zero, or one. in this scenario we are extending a twenty six bit immediate to a thritytwo bit value

4. a Jump-type instruction interpreting the immediate field as unsigned: This scenario is not implemented in the datapath as the jump instructions always consider a signed value of the immediate field. However for a future improvement of the DLX it could be used for a type of jump instruction that can only jump forward.

All these scenarios are performed in parallel in the decode unit, and thanks to a four to one multiplexer, the correct specific scenario is selected for obtaining the appropriate extension. Furthermore, the selection bits of the multiplexer individuating the correct extension scenario come from two control bits from the control unit.

4.2 Register file

The Register file of the DLX consists of thirtytwo registers. The register at address zero is always zero by convetion while the thirtysecond register is only utlized for the storing of the return address when executing a jump and link (jal) instruction. A main schematic block is shown in Figure 4.1 .

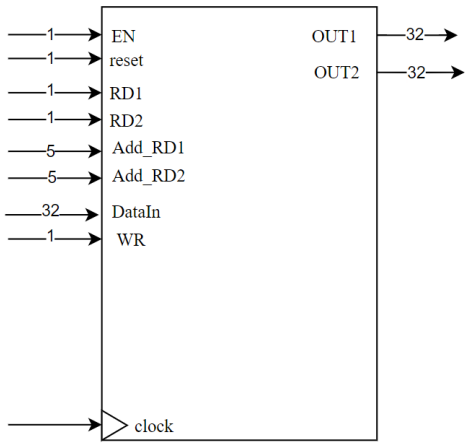


Figure 4.1: Register File Schematic

4.2.1 Reading Methodology

The register file has two read ports. The reading is done asynchronously this way it is easier for the register file to obtain the proper addresses of the register for which the reading is performed. On the basis of the RD1 and RD2 single bit port signals, the ports for reading can be activated. After receiving the proper addresses from the Instruction register bits, the values of the registers are made available at the OUT1 and OUT2 ports respectively.

4.2.2 Writing Methodology

The writing methodology is synchronous to falling edges of the clock cycle. This is done to avoid the issue of metastability.

In fact if the register file were to sample the DataIn port at the rising edge of the clock cycle, since the control unit sets the write port at rising edges of the clock cycle, the register file would understand as if the write port bit is zero hence leading to an incurrect behavior. Furthermore, this is the reason why the write process is sensitive to the falling edge od the clock. About the reset, it is sensistive to the falling edge of the clock and when it is active high. At the start up of the processor, thanks to the reset all the registers of the register file as set to inital value zero.

4.3 Decode Pipe Registers

There are many Pipeline registers which store relative information all of them having syncronous enabling and resetting at the rising edge of the clock cycle:

1. NPC1: This pipeline register is needed as it needs to be propagated to the Execution unit in case there is the need to compute an address because the instruction of interest is a branch or a jump.

2. RegisterA and RegisterB: These registers store the output of what was read from the asynchronous reading of the register file. If in the decode stage there is a Register type instruction both of these registers will be filled with useful values to be later used in the Execute stage.

3. ImmREG : This register stores the final extended form of the immediate value that was chosed from the four to 1 multiplexer described earlier in this section.

4. RT and RD Registers: the need of this register is important do the fact that there is a variation in position of the destination register field of an instruction; in fact for immediate type instruction the destination register address is stored in bit field range [20 downto 16]. While for the Register type instruction it is located in the bit range [15 downto 11] of the instruction. Hence in the decode both information are stored, to then decide based on the instruciton type which is the correct address for writing back to the register file.

5. IR1: This will keep the instruction that was analyzed during the decode stage also in execute. This register was implemented for viewing the waveforms while debugging the processor's course of actions.

CHAPTER 5

Execution Stage

After the Decode stage there is the Execution unit which consists of computation of the acquired information during the earlier decode stage. This part of the datapath has a very important component called the Arithmetic Logic Unit which is utilized for perforing various operations depending on the specific instuction. This part of the datapath also contains the important digital circuitry used for analyzing branches and jumps, needed to understand what address to place at the content of the program counter in the fetch phase for the next clock cycle. Lastly a brief description of the pipeline registers related to this stage is provided.

5.1 The Arithmetic Logic Unit

The DLX has these following arithmetic operations:

1. T2 Shifter;
2. Pentium 4 Adder;
3. Comparator ;
4. T2 Logical ;

Furthermore Figure 5.1 depicts the black box structure of the Arithmetic Logic Unit. There is a final four to 1 multiplexer which selects which results needs to be given as the ouput of the operation. This selection depends on the particular instruction's predefined control bits that are given as output from the control unit. The operands A and B from which the ALU performs the introduced operations are finalized thanks to two independent multiplexers. One multiplexer chooses if the Operand called Operand A needs to feedthrough the Next Program Counter Value, or the Value which came from OUT1 port of the register file. On the other side, the second multiplexer chooses if the second operand called operand B should be the value read from port OUT2 of the register file, or the Immediate field. There are 2 control bits which make the decision as they are the selection signals of these multiplexers.

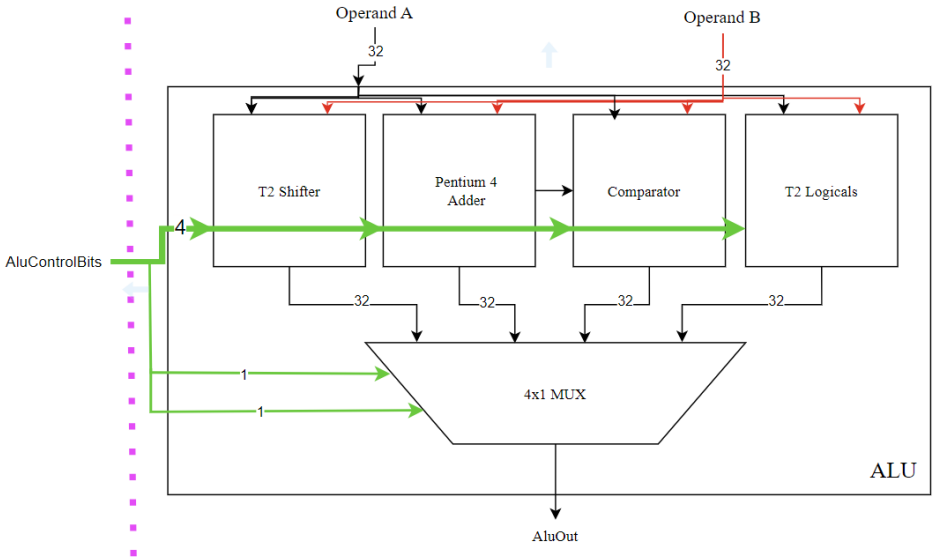


Figure 5.1: Schematic of internal Black Box ALU components

5.1.1 T2 Shifter

The shifting methodology used is the same one used as the T2. This means that the shifting is performed on the basis of the following three levels:

1. First Level: formation of all the masks that are to be applied in the next levels of the computation
2. Second Level: on the basis of the formed masks during the first level and the amount of shifting that is to be done, the mask which is most similar to the real amount that needs to be shifted is selected. Alternatevely it can be said that this level is broad shifting operation
3. Third Level: starting from the output of the shifted version from the Second Level of computation a fine grained level shifting is done at this level, which means that the final shifted result of the shifting operation is finally calculated.

Figure 5.2 rapresents the concepts that are above discussed (all though applied assuming that the input is of 64 bits).

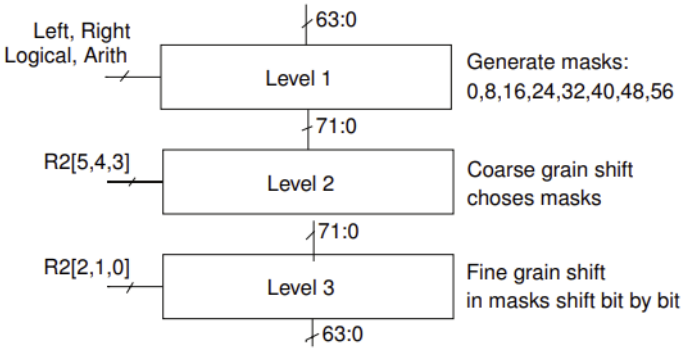


Figure 5.2: Black Box schematic of the T2 Shifter

5.1.2 The Pentium 4 Adder

It is an efficient adder whose implementation is to minimize the critical path by making the bits of the operands that are being added or subtracted less vulnerable on the carry bit propagation during the addition. Specifically, the adder is composed of two main blocks:

1. The Carry Look Ahead Sparse Tree: It is a tree like structure formed of interconnected Propagate and Propagate-Generate blocks included for relying less on the dependency of the carry out done formed in a previous stages, for reduction of the critial path.

2. The Carry Select Sum Block: similar to the Carry Look Ahead Adder concept; it is formed of a series of stages made of a specific predefined amount of concatenated full adders. Each stage has two parrallel chains of full adders which perform respectively the addition on the hypothesis that the carry in bit is zero and one respectively. Then, the correct output of the addition between a particular subset of bits of the operands is chosen on the basis of the actual local carry produced by the sparse tree block described previously.

Summarizing, the adder/ subtractor module is implemented in a more efficient manner this way do to the fact that the addition of the particular subset of bits of the operands are all done in parallel unlike for instance the basic ripple carry adder model. Additionally, each set of ripple carry adders in the Carry Select Sum Block does not have to wait for the carry out from the previous stage to perform the computation. Meaning that thanks to the Sparse tree, the addition/subtraction of the Pentium4 adder is more efficiently parallelized with respect to the addition performed by the Carry Look Ahead Adder model.

Figure 5.3 depicts the block schematic of the described high level blocks which make up the pentium 4 adder module.

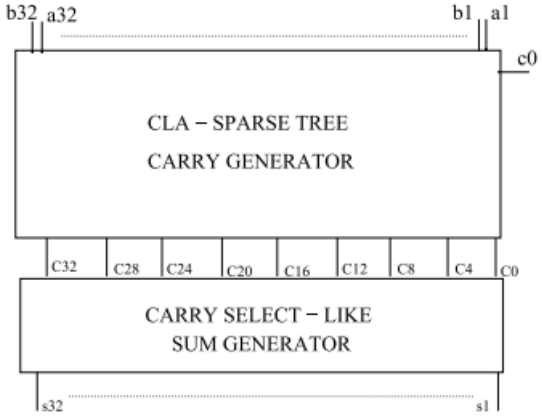


Figure 5.3: Black Box schematic of the Pentium 4 adder.

5.1.3 The Comparator

The implemented comparator is constructed using the ouput of the present pentium four adder. Moreover, if two operands need to be compared, they will be firstly given to the pentium 4 block which will perform a subtraction. The comparator block will analize the carry out obtained from the results along with a flag bit called Z used set to one if the result of the subtraction is zero. Additionally the implemented comparator is able to perform a dual kind of operation in case the operands are treated as unisgned instead of signed. Figure 5.4 reports the schematic depicting the implementation of the comparator for the operands which are understood as signed.

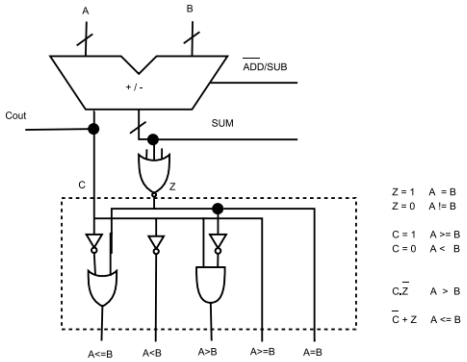


Figure 5.4: The Comparator

5.1.4 The T2 Logicals

Logical instructions like the AND, OR and many others are performed through the T2 circuitry design. This choice was made as it leads to the implementation of a lot of logical operations still while using few gates (meaning less transistors equivalently). The structure is composed of four and gates in the fisrt level of the hirierchy. Then, the second level is one additional final and gate doing the and between the outputs of the first level. On the basis of the control bits of the dedicated to the execute phase there is the possibility to implement the logical operation of interest. Figure 5.5 depicts the circuit methodology of the comparator for the situation in which the .

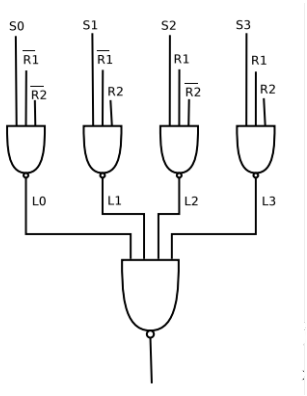
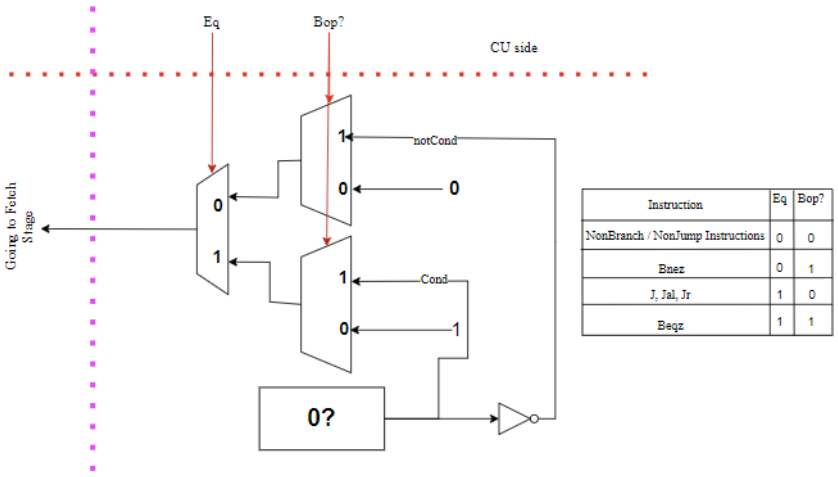


Figure 5.5: The T2 Circuit for Logical Operations

For more details on the components specified above, chapter four from the Microelectronic Systems Lecture notes provides a more thorough analysis.

5.2 Branch and Jump Related Circuitry

The DLX sends out the selection signal for jumping in the fetch stage in the execution stage. Similarly, in case of a branch instruction, the DLX determines whether the branch needs to be taken or not in the execution stage. the mechanism implemented for knowing if the branch and jump need to be taken is composed of three multiplexers distributed hirierchically. In the first level of the hirierchy, there are two multiplexers. one of them receives 1 and a signal called 'cond' (wich is the flag indicating if value stored the "REGA" pipe decode register is zero or not), and the other multiplexer has zero as first input and a signal called 'notCondIn' (which is the inversion of the of the CondIn signal). The output of these two multiplexers are controled by one bit of the control unit which is one if the instruction is a Branch instruction, while when it is zero the instruction at analysis is not a branch. In the second level of the hiriechy of the multiplexers there is one multiplexer which decides how to interpret the input it has (if it needs for instance to branch due to an inequality of zero or due to an equality of zero.) Also for this second level of the multiplexer hiriechy there is a dedicated control bit from the control unit. Figure 5.6 depicts the multiplexer hiriechy that was discussed along with a table associating the control bits of interest to the various intructions



Instruction	Eq	Bop?
NonBranch / NonJump Instructions	0	0
Bnez	0	1
J, Jal, Jr	1	0
Beqz	1	1

Figure 5.6: The two level hirierchy of the multiplexers with an explenation table

5.3 Execution Pipe Registers

The execution stage has the following Pipe Registers:

1. ALUOutReg: in this pipe register the result of the alu operation is stored in the case that for instance the result needs to be written back to the register file during the write Back Stage

2. NPC2: this pipe register is needed for propagating the NPC address in the eventual case that the instruction of interest is a jump and link instruction. In such case there will be the need to store this address in the the thirtysecond address of the register file during the write back, hence the reason we need to save it.

3. ForMemStRegister: this register is needed in the case the instruction is of store type. Thanks to this register the subsequent stage (the memory stage) will be able to obtain the correct value to store in the external DRAM memory.

4. RDdestReg: In the execution stage the final decision on which is the correct destination address for writing is made (between rd or rt depending on if the instruction is immediate or register type). Hence this register is needed to store this finalized address so that during the future Write Back stage, the writing of the result is performed in the correct location of the register file

5. IR2: needed for help during the debug process while checking the waves on Modelsim while testing the single datapath unit.

CHAPTER 6

Memory Stage

Memory Unit is the fourth stage of the pipeline. It is composed by the Data Memory and the pipeline registers. There are five different registers:

1. Three out of five are used to pipeline the destination address, the npc and the instruction.
2. One is used to pipeline the result of the ALU from the execution stage.
3. The last one is used to take the output of the Dram in the memory stage in case there is the need to write it to the register file (load instruction).

6.1 Dram

The Dram contains the data for load/store instructions and the values available at the start of the execution are initialized during the reset phase thanks to "DMEM_init_file.mem". This way, initial values are loaded into the DRAM at startup which can be a helpful addition during the testing of the store and load instructions in Modelsim The correct instruction is executed based on RW signal(read/write signal) that it is sent by the Control Unit.

6.1.1 Read and Write processes

During the process, as inputs arrive an unique address, the data pipelined from the execution unit (RegB), RW and EN from the Control Unit. The read operation starts when EN='1' and RW='1' (load instruction) instead for write operation when EN='1' and RW='0' (store instruction), both synchronous with the falling edge of the clock. The output ,in case of the read operation, is sent to the pipeline register LMDreg in memory unit to be pipelined to writeback.

CHAPTER 7

Writeback Stage

Writeback Unit is the fifth stage of the pipeline and it is composed by three different multiplexers, controlled, using SEL signals, by three different bits of the control word sent by the CU.

1. The first one takes as inputs the output of the alu pipeline register and the output of the dram pipeline register. This mux is used to choose between a value obtained from the DRAM with controlwordbit1='1' (load instruction) or from the ALU with controlword1='0'.

2. The second takes as inputs the output of the first multiplexer and the output of the NPC3 pipeline register. In this case , it is used to choose which value must be saved in the register file, NPC3 with controlwordbit2='1' in case of jal instruction or the value from the previous mux controlword2='0'.

3. The last one is linked to the second mux because it chooses 31 as value when it is a jal and so the value of NPC3 must be stored in r31 with controlword3='1', in the other cases the value of the address destination pipeline register with controlwordbit3='0';

CHAPTER 8

Control Unit

The Control Unit is an Hardwired Control Unit and controls the Datapath generating the proper controlword based on current instruction in the proper clock cycle. It is divided in different parts:

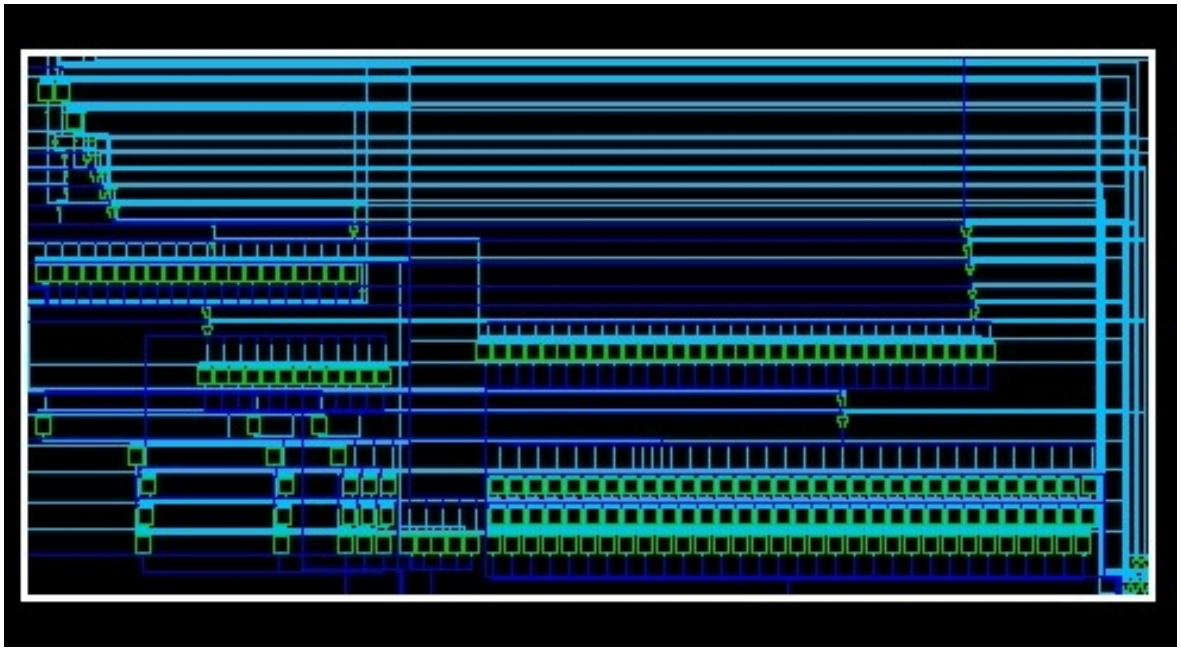


Figure 8.1: Synthetized Control Unit

1. Look-Up Table: it contains all the control words linked to the instrunctions
2. Three processes that work one after the other:
   (a) The first one takes the main input of CU (IR_In) and consider the relative opcode and func of the instruction which are used by the second process.
   (b) The second one takes the opcode and func of the first process (checking if RTYPE,ITYPE,JTYPE) and use them to take the correct control word from the look-up table.
   (c) The last one works to pipeline the control words in the right way (as an Hardwired Control Unit do) and there is also the pipeline of the instructions that it is used for the Hazard Unit.This process, also, takes as inputs the three hazard signals from the HU to flush the

pipeline putting NOP in the decode stage and execute stage for branch hazard or putting NOP in decode stage for jump untill the new PC or putting NOP in execute stage and maintaining the same control word (and istruction)in the decode stage for raw hazard.The flush of the pipeline depends on where the hazard is taken (specified in the HU chapter), it lasts one or more cycles.

8.0.1 Control Word

The main job of the Control Unit is to generate the right control word using the look-up table. Each control word is saved in the file "INSTR_CODES.vhd" with all the opcode and func of the instructions. The bits are divided for Decode Unit, Execution Unit, Memory Unit, Writeback Unit and they are pipelined during the cycles to be used in the correct moment of the pipeline to activate the useful components of the datapath.

CHAPTER 9

Hazard Unit

The Hazard Unit is used to detect 3 different type of hazard:

1. Raw Hazard in which the pipeline stall to wait the value to read for the current instruction stalled
2. Jump Hazard when a jump is in decode and flush the instructions after the jump, waiting the new PC
3. Branch Hazard when a branch is taken in execution unit , flush the pipeline and put new PC

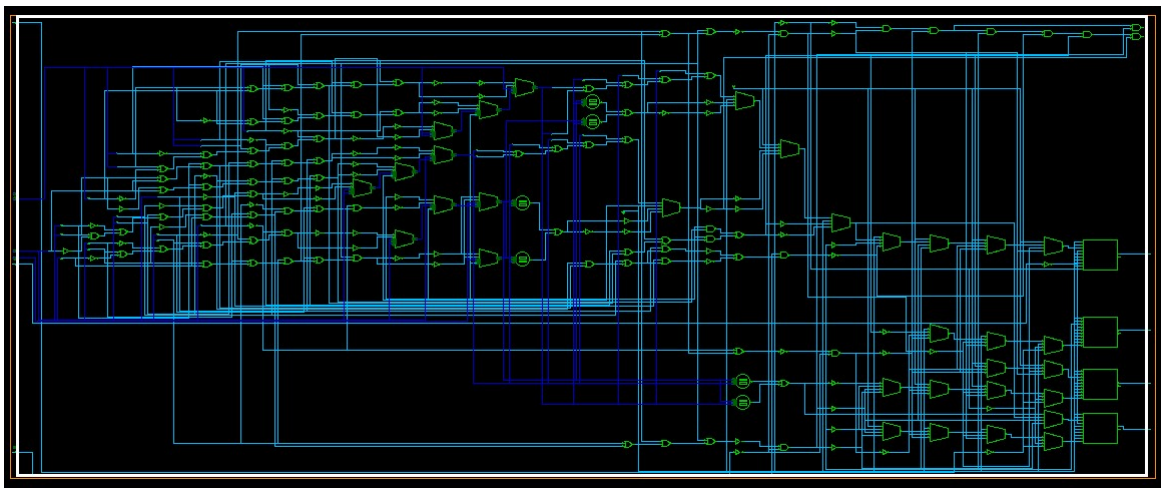


Figure 9.1: HU

To detect these hazards, the HU works on two processes, executed one after the other.

1. First process: It works on the inputs coming from the CU (IR_ID,IR_EX,IR_MEM). It uses the pipeline and the instruction with its relative opcode (ITYPE,JTYPE,RTYPE) to take the correct registers used (RS1,RS2,RD).This analysis creates also a pipeline of the registers to take into account in which stage the register is, to consider in a correct way raw hazards.
2. Second process: It is the process used to detect the hazard and it is divided in 2 main part:
   (a) In the first part simply there is the detection of a jump/jr/jal in the decode stage (for jr there is also a control on the register used that it could create raw hazard).Also the

detection of the branch, in the execute stage, thanks to the result (Branchstatus) given by comp4Branch in the Datapath.In the ifs of jump/jr/jal there is an "AND branchstatus" because of a problem caused by a branch followed by a jump that created an error in which of the two was the correct to take.

(b) Second part: It is used to detect raw hazards between the destination registers in execution-memory with the source registers of the instruction in decode (using the pipeline registers explained before). All the possible combination are considered (ITYPE,RTYPE different source registers and destination registers positions)

When a hazard is detected a signal called PC_sel is rised to stop the pc (except for branch) and stalls are implemented in the pipeline with the insertion of NOPs in the correct stages (explained in CU).After the useful stalls, the hazard signal and the PC_sel go at 0 and the code restarts its correct pipeline.

CHAPTER 10

Testing

10.1 Simulation

The simulation process has the purpose of verifying the correct behavior of the DLX. The method used to achieve this result was the usage of testbenches. First step was to create testbenches for the internal modules, such as the P4 Adder, the ALU, the T2 shifter and so on. Using a bottom-up approach, the correctness of the internal modules was confirmed and the testing was moved on higher levels, up to the entire DLX.

The same structure was used for all the testbenches, which were all written using VHDL. They all have one process, which generates a clock signal (1 ns was used as a reference clock), and another process which provides input to the tested unit, asserts reset and enable signal, in order to make the unit to provide outputs, which are verified using the *Modelsim* simulator, that provides a nice GUI to visualize the waveforms.

The testing of the whole DLX consisted in loading the bytes of an assembly program to a file, which is read by the Instruction Memory (IRAM). Another file is used to eventually initialize the DRAM, which can be useful in presence of load instructions, for example. Then, after asserting the reset signal, the program starts executing until there are no more instructions to be read. A script which acts as an assembler is used to read an assembly program and translate it into a sequence of bytes (which are then loaded into memory). A file called *sim_howto.txt* shows the steps to run a complete simulation using Modelsim. Here, a waveform is shown, that shows the signals and behavior of a program.

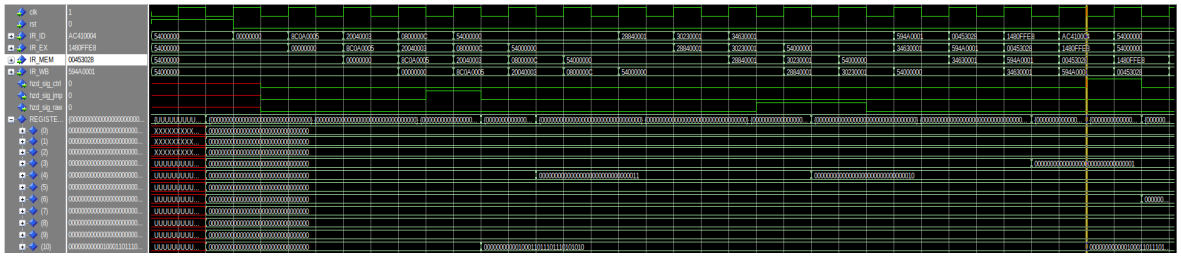


Figure 10.1: Modelsim waveform

CHAPTER 11

Synthesis

The synthesis process was done after the Simulation phase, in which the design was simulated to ensure correct behavior of the different units of the CPU, and the CPU itself. The usage of scripts to automate the synthesis steps is crucial, since it allows us to produce the design netlists and reports saving a lot of time.

The script synth.sh was created to automate the synthesis and optimization of the CPU under timing and power constraints. It is a little bash script which starts the synopsys shell in order to perform the synthesis. The paths to the vhdl modules to be analyzed are written in sources.f, which is read by synthesis.do, which is another scripts that execute all the synthesis steps, once the synopsys shell is opened.

Various steps are performed in this phase, from generating detailed models of the design, with timing, area and power reports, to implement optimizations with regard to timing constraints, for example.

The whole process is composed of multiple steps:

1. First, all vhd files needed for the DLX synthesis are analyzed (testbenches are not used in synthesis)
2. The design is then elaborated and high level modeles are produced. At this point, at this point we don't have many information about area, timing and power consumption.
3. The wire model is set and the clock signal is created
4. The design is compiled and the netlists are generated, ready to be analyzed by a physical design tool.
5. The timing, area and power reports are saved.

The timing report is especially important to see if the chosen clock period value is suitable for this design. The report shows the critical path(s) of the design and tells if the **slack** is met.

This DLX design contains different optimizations with regard to combinational circuits. The P4 Adder, created during the course labs, was used in this design. The T2 shifter is another addition to the design, which provides speed advantage to shifting operations.

Since the finding of the right clock period was a sort of trial and error process, the timing report was used to see if some other optimizations were possible. The first timing report shown a critical path in a RCA (Ripple Carry Adder) module, which was used in the incrementing of the program counter. The RCA was implemented using a structural design, which provided a lot of delay, due to the carry propagation in the circuit. We switched to a behavioral implementation, letting the synthesis libraries

do the job for the optimal netlist. The timing performance was substantially better and we managed to get a clock period of 1.50 ns for the whole DLX. At this point the shown critical path is a series of gates which belong to the hazard unit. Since the provided Hazard Unit has a high level behavioral implementation, it was assumed that we could not optimize very much at a high level of abstraction.

Analyzing the other reports, area and power information can be provided. The DLX has a total cell area of 19141.89, which is divided between combinational area (8752.19) and non-combinational area (10389.69). The total dynamic power is 818.39 mW, with a leakage power of 399.45 uW. Here is shown the Area report, the timing and power report are in appendix A1.

```
****************************************
Report : area
Design : CPU
Version: S-2021.06-SP4
Date   : Wed Oct 18 21:01:51 2023
****************************************

Number of ports:                         4781
Number of nets:                         15770
Number of cells:                        10438
Number of combinational cells:           7909
Number of sequential cells:              2300
Number of macros/black boxes:               0
Number of buf/inv:                       2311
Number of references:                       3

Combinational area:               8752.198124
Buf/Inv area:                     1476.299987
Noncombinational area:           10389.693676
Macro/Black Box area:               0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 19141.891799
Total area:                 undefined
1
```

CHAPTER 12

Physical Design

This is the final step, in which the layout of our DLX design can be generated, using the **Innovus** tool, provided by Cadence. The entire procedure is composed of various steps, from Floorplanning to Cell Placing and signal Routing.

To start creating the floorplan, Innovus needs a verilog post-synthesis netlist, a *.lef* file that has references to the library of cells and other data which is provided in the project files. In this step the area dedicated to the chip core is computed, along with the power supply rings around it. Next, the power rings for *Vdd* and *GND* are inserted in the model and on the layout corner, the vias to connect the metal layer are placed as well. To avoid congestion, M9 and M10 metals are chosen for the rings. The rings are then connected using vertical stripes and horizontal lines, to better distribute power into the center of the chip. Power Grid Placement is shown in Figure 12.1.

12.1 Placement and Routing

In the Placement phase, the cells are placed, along with I/O pins. Post CTS (Clock-Tree-Synthesis) optimization is performed before routing. The result is shown in Figure 12.2. Before the routing phase, it is important to complete the placement filling free spaces in the layout with *Filler Cells*, to maintain the continuity of the N+ and P+ wells.

The final step is the **Post routing optimization**, in which the design is optimized in order to achieve the timing constraints. Now the design is complete and a final verification can be performed to see if there are violations with respect to design rules and connectivity.

The final layout is shown in 12.3

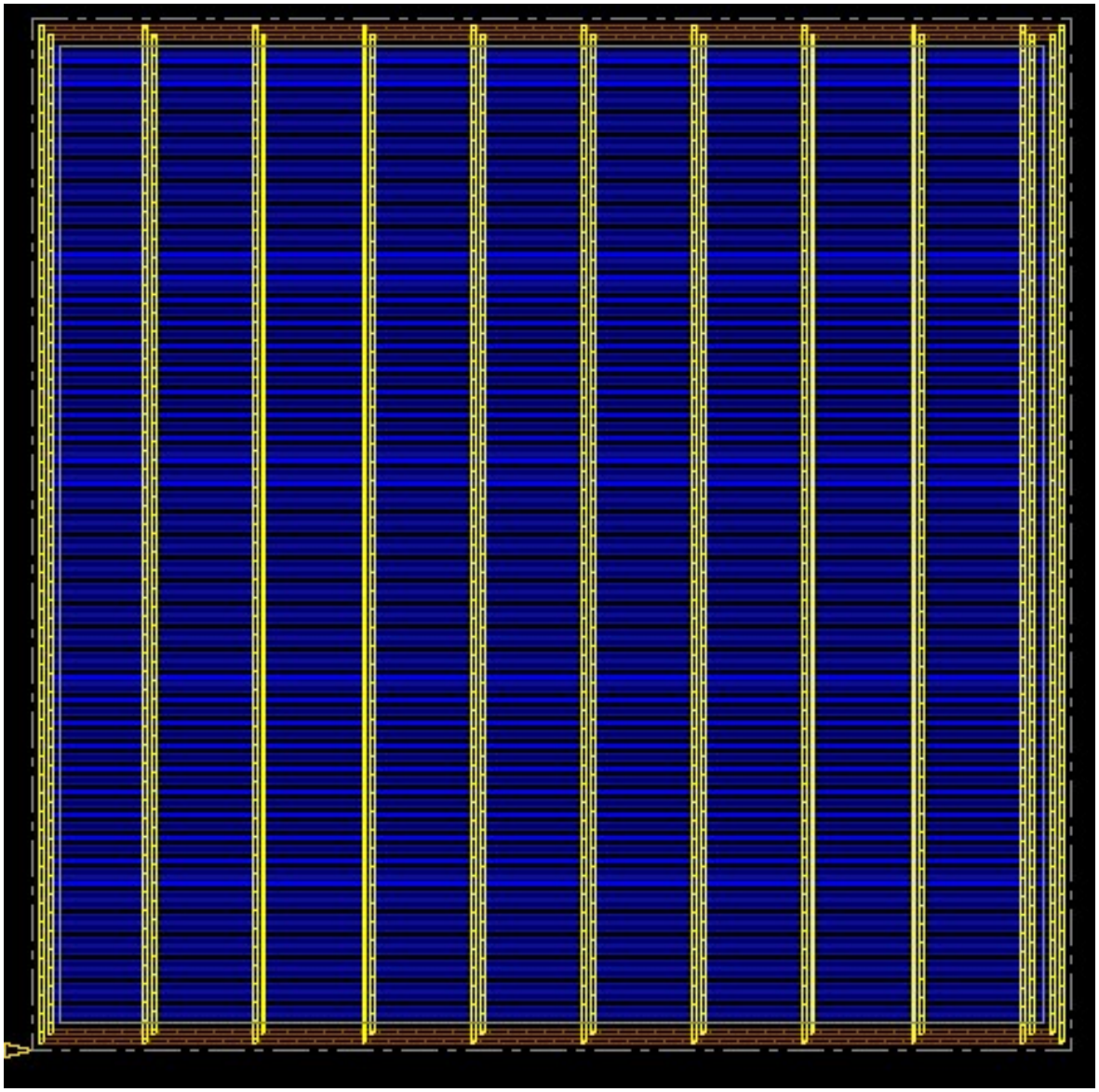


Figure 12.1: Power grid distribution

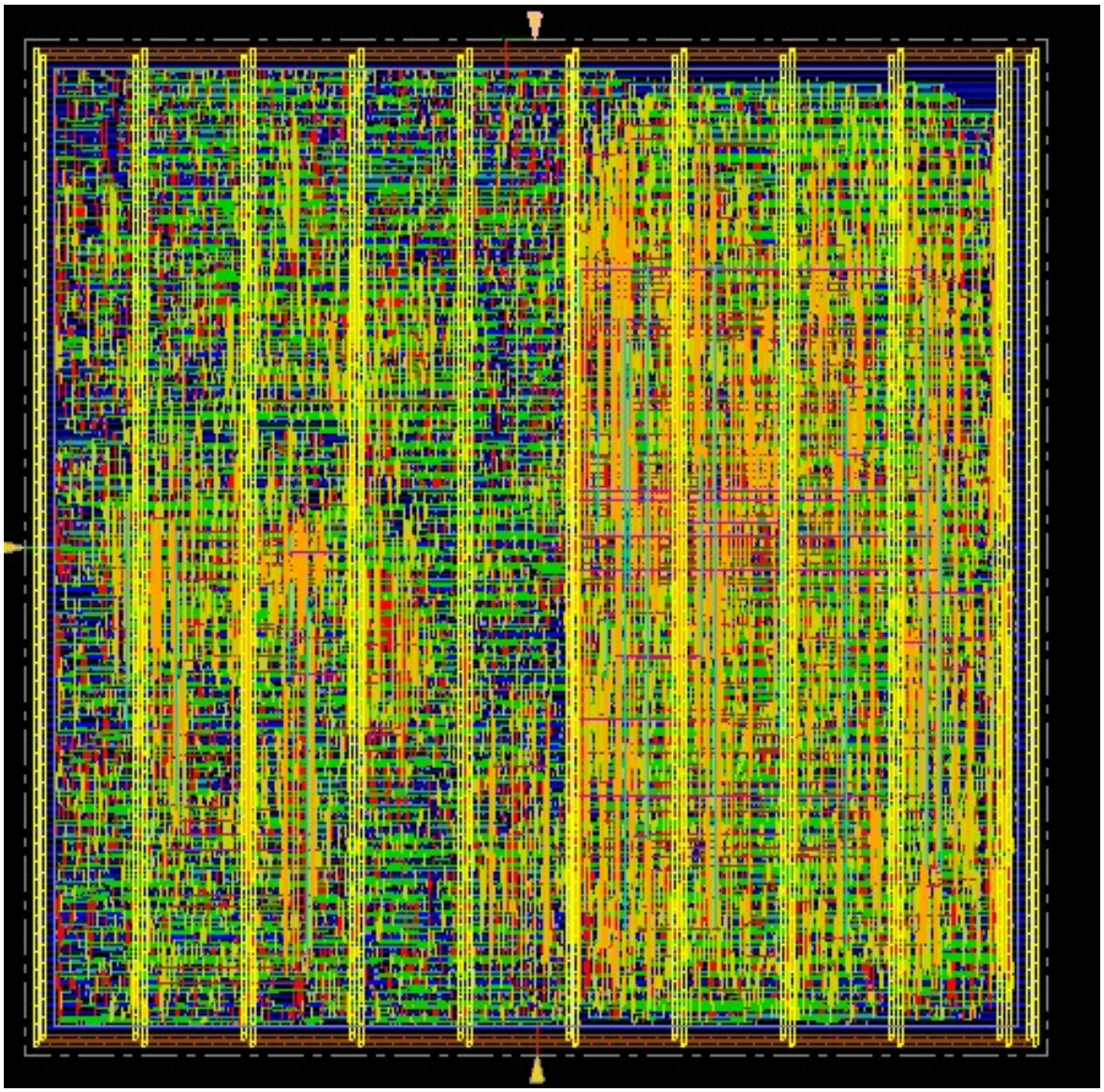


Figure 12.2: Post CTS Optimization

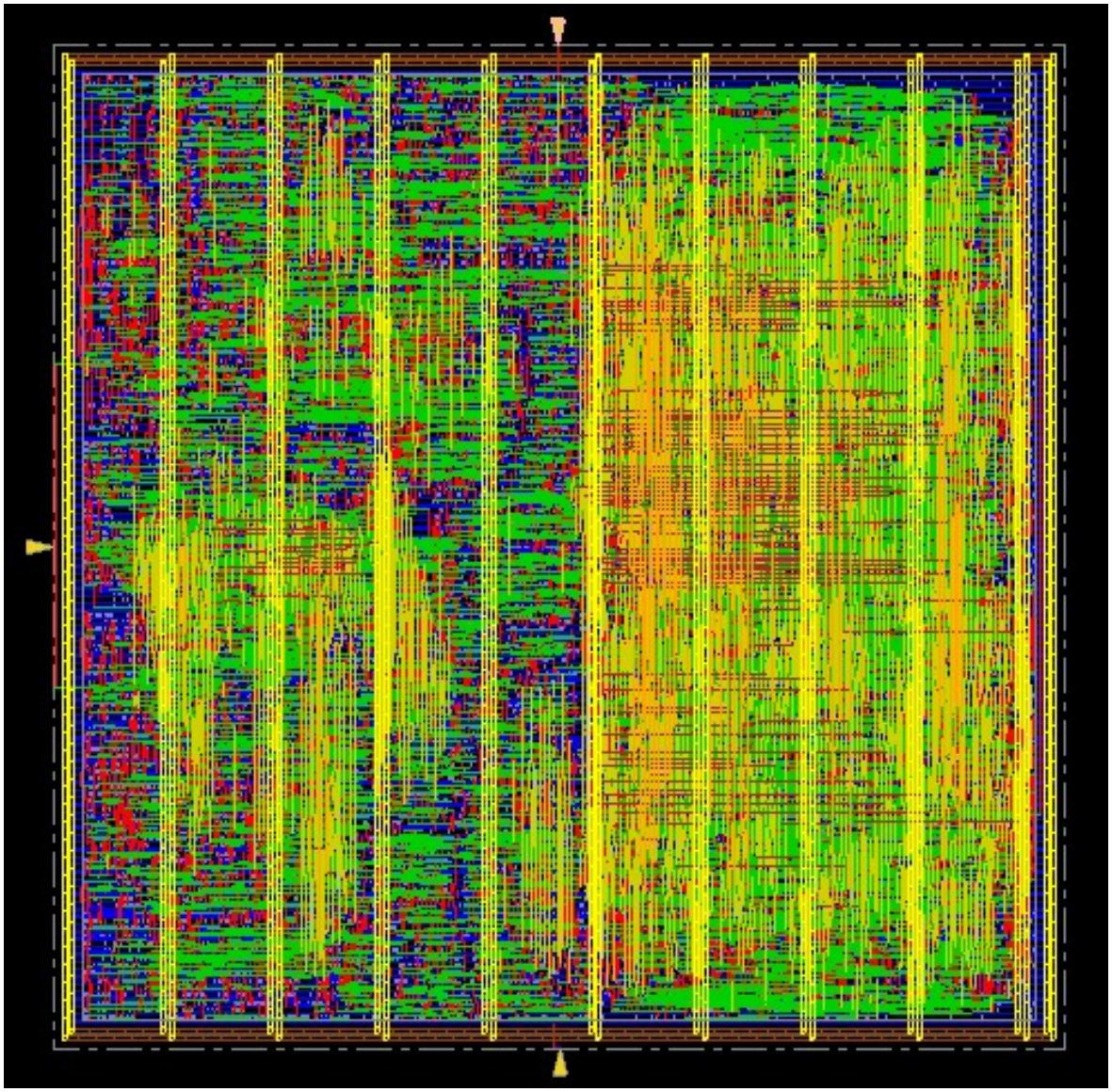


Figure 12.3: Final layout

APPENDIX A

Synthesis Reports

A.1 Power Report

```
****************************************
Report : power
        -analysis_effort low
Design : CPU
Version: S-2021.06-SP4
Date   : Wed Oct 18 21:01:54 2023
****************************************

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

Design          Wire Load Model             Library
------------------------------------------------
CPU                     5K_hvratio_1_4      NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =  10.3023 mW    (1%)
  Net Switching Power  = 808.0876 mW   (99%)
                         ---------
Total Dynamic Power    = 818.3898 mW  (100%)

Cell Leakage Power     = 399.4569 uW
```

Power Group	Internal Power	Switching Power	Leakage Power
io_pad	0.0000	0.0000	0.0000
memory	0.0000	0.0000	0.0000
black_box	0.0000	0.0000	0.0000
clock_network	807.2925	8.0739e+05	43.0596
register	9.2661e+03	60.0180	1.7313e+05
sequential	0.0000	0.0000	0.0000
combinational	228.8129	642.4100	2.2628e+05
Total	1.0302e+04 uW	8.0809e+05 uW	3.9946e+05 nW

Total Power (%)	Attrs
0.0000 (0.00%)	
0.0000 (0.00%)	
0.0000 (0.00%)	
8.0819e+05 (98.71%)	
9.4993e+03 (1.16%)	
0.0000 (0.00%)	
1.0975e+03 (0.13%)	
8.1879e+05 uW	

A.2 Timing Report

```
Information: Updating design information... (UID-85)
Warning: Design 'CPU' contains 2 high-fanout nets.
A fanout number of 1000 will be used for delay calculations
 involving these nets. (TIM-134)

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : CPU
Version: S-2021.06-SP4
Date   : Wed Oct 18 21:01:51 2023
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top
```

```
Startpoint: CU/IR_EX_s_reg[31]
            (rising edge-triggered flip-flop clocked by CLK)
Endpoint: HU1/hzd_sig_raw_reg
          (rising edge-triggered flip-flop clocked by CLK')
Path Group: CLK
Path Type: max
```

Des/Clust/Port	Wire Load Model	Library
CPU	5K_hvratio_1_4	NangateOpenCellLibrary

Point		Incr		Path
clock CLK (rise edge)		0.00	0.00	
clock network delay (ideal)		0.00	0.00	
CU/IR_EX_s_reg[31]/CK (SDFFR_X1)	#	0.00	0.00	r
CU/IR_EX_s_reg[31]/Q (SDFFR_X1)		0.08	0.08	f
CU/IR_EX[31]				
		0.00	0.08	f
HU1/IR_EX[31] (HU)		0.00	0.08	f
HU1/U136/ZN (NOR3_X1)		0.09	0.17	r
HU1/U102/ZN (AND4_X1)		0.08	0.25	r
HU1/U158/ZN (NOR3_X1)		0.03	0.28	f
HU1/U147/ZN (AND2_X1)		0.05	0.32	f
HU1/U148/ZN (NOR3_X1)		0.08	0.40	r
HU1/U139/ZN (XNOR2_X1)		0.04	0.44	f
HU1/U138/ZN (AND2_X1)		0.04	0.49	f
HU1/U179/ZN (NAND4_X1)		0.03	0.52	r
HU1/U52/ZN (AOI21_X1)		0.03	0.55	f
HU1/U161/ZN (OAI21_X1)		0.05	0.60	r
HU1/U168/ZN (NAND2_X1)		0.03	0.63	f
HU1/U167/Z (MUX2_X1)		0.07	0.70	f
HU1/hzd_sig_raw_reg/D (DFF_X1)		0.01	0.71	f
data arrival time			0.71	
clock CLK' (rise edge)		0.75	0.75	
clock network delay (ideal)		0.00	0.75	
HU1/hzd_sig_raw_reg/CK (DFF_X1)		0.00	0.75	r
library setup time		-0.04	0.71	
data required time			0.71	
data required time			0.71	
data arrival time			-0.71	
slack (MET)			0.00	