TEAM LILAC

LILAK

ASSEMBLY ARCHITECTURE

LILAKILLER
MICROPROCESSER

LILAK
ASSEMBLY ARCHITECTURE

LILAKILLER
MICROPROCESSER

TABLE OF CONTENTS

LILAK

ASSEMBLY ARCHITECTURE

LILAKILLER MICROPROCESSER

DESCRIPTION

LilaK (LilaKILLER) is an assembly based processor with a load-store architecture that utilizes a minimalist core instruction set and a majority of pseudocode implementation to simplify the user experience and allow for better understanding of a program. The language provides an easy-to-read user interface through descriptive instruction calls for new or less experienced programmers.

The processor uses a 16-bit address bus and a 16-bit data bus to execute programs stored in an external memory location and is able to take in basic, 16-bit inputs and output the resulting data of internal computations. LilaK can be used in cases of general computations and supports parameterized and nested procedures.

FEATURES

The LilaKiller microprocessor has a few of these defining features:

- Five-stage pipeline architecture.
- Considerable collection of pseudoinstructions.
- Fully-functional Python assembler and compiler.

APPLICATIONS

The LilaK microprocessor has the following applications:

- Aid in student-learning about machines & machine code.
- Run small, simple programs including, but not limited to:
  - Loops
  - Parametrized functions

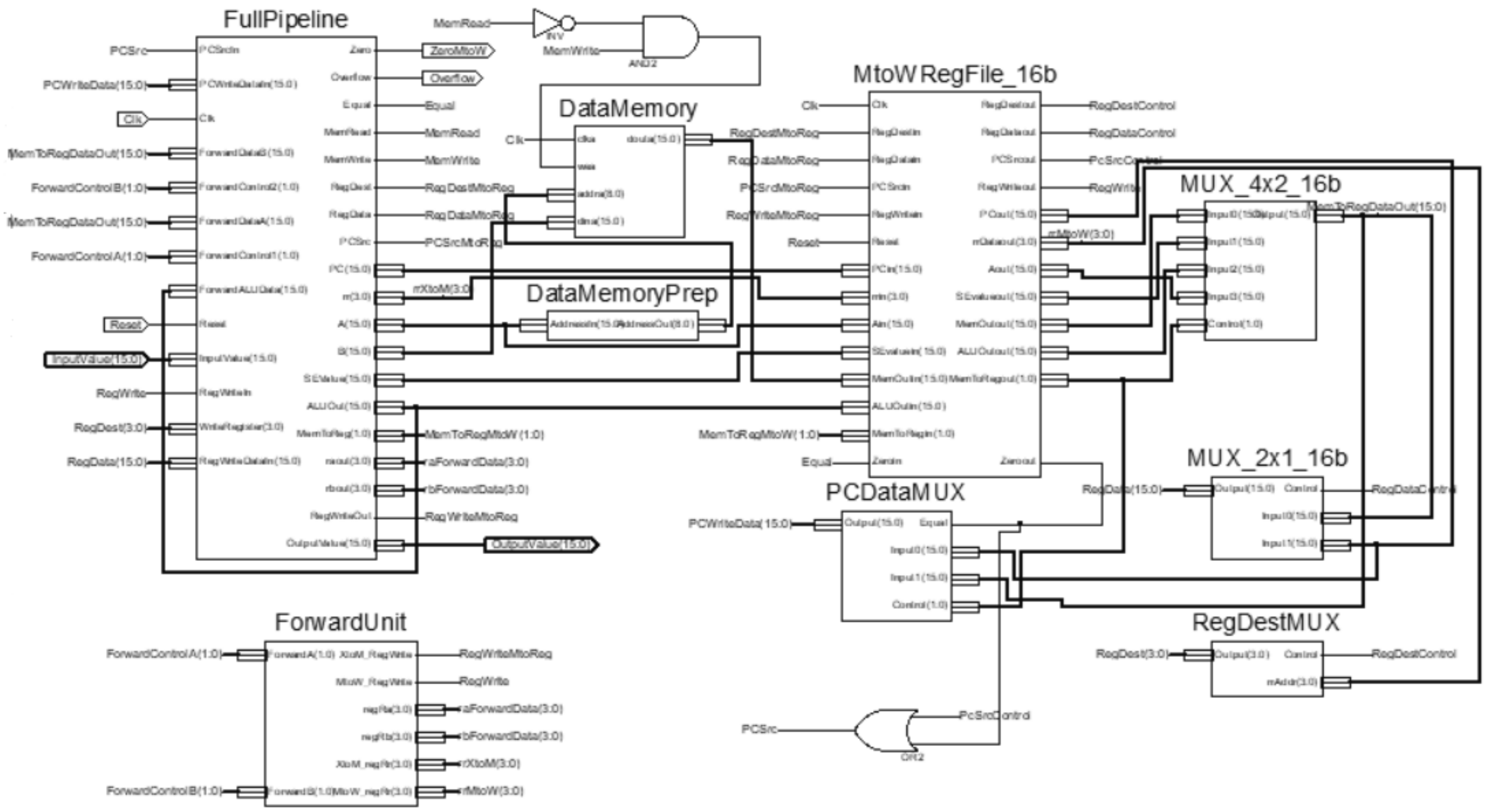


Figure 1: Full LilaK Circuit (Xilinx)

OVERVIEW

Lilak is intended for use by new or less experienced programmers. The language is similar to other load and store architectures such as MIPS and ARM, but has additional implemented features and design goals. The easy-to-read and simple instruction set and pseudoinstructions along with a very involved and sophisticated assembler and compiler program allows the programmer to just focus on and learn with the program in front of them. The pipelined, load and store architecture is intended to shorten the necessary processing time and to save space in the data path.

LILAK
ASSEMBLY ARCHITECTURE

LILAKILLER
MICROPROCESSER

DATA FORMAT

The LilaKiller processor defines a 16-bit word, and 8-bit halfwords. The byte ordering is Little-Endian, with the most significant bit being stored at the highest address in memory.

PROCESSOR RESOURCES

LilaKiller uses sixteen 16-bit data registers. Eleven of these registers are intended to be used by the programmer. The remaining seven are only to be used by the assembler & memory operations outside of the scope of the programmer. The register list is found in **Figure 2.**

REGISTER NUMBER	REGISTER NAME	REGISTER DESCRIPTION	TO BE PRESERVED ACROSS A CALL?
0	$zero	Hardcoded Zero	N/A
1	$ra	Return Address	Yes
2	$stack	Stack Pointer	Yes
3	$global	Global Pointer	Yes
4	$frame	Frame Pointer	Yes
5	$in	Input Register	No
6	$a0	Assembler Use Regs.	No
7-8	$fa0, $fa1	Procedure Arg. Regs.	No
9-10	$fr0, $fr1	Procedure Return Regs.	No
11-12	$v0, $v1	Temp. Value Regs	No
13-15	$sv0 - $sv2	Saved Temp. Value Regs.	Yes

Figure 2: Register Table

The LilaK processor utilizes PC-relative addressing for all operations, with the exception of jump instructions that utilize PC-direct addressing.

INSTRUCTION TYPES

Despite the LilaK processor only having two instruction sets, the processor features a surplus of beginner and intermediate instructions that fit within the boundaries of the types.

Load & Store instructions are how LilaK moves data to and from memory and the programmer-accessible registers. These instructions are of the A-type.

Computational instructions are how LilaK performs arithmetic, logical, and other general mathematical instructions like multiplication and division. These operations occur on value registers, but the pseudo-instruction-heavy design of LilaK allows the programmer to do computations using immediate values as well. All computational instructions are A-type.

Jump & Branch instructions are how LilaK changes the location of the program counter in the code. LilaK jump instructions utilize PC-direct addressing, whereas branching instructions use PC-relative addressing. Both jump and branch instructions are still A-types.

Set instructions are the only V-types of the LilaK assembly language. Set allows the programmer to directly set the result register to a specified immediate value.

Psuedo instructions are the bread and butter of the LilaK processor. With the intention of beginner programmer use and an ease into learning machine language, the goal of LilaK's heavy pseudo instruction arsenal is to help bridge an understanding between human language and machine language. Such pseudo instructions involving values, like addval & subtractval, utilize the assembler registers to allow addition and subtraction of immediate values.

A-TYPE ARITHMETIC & LOGIC

OP	ra	rb	rr
4 bits	4 bits	4 bits	4 bits

V-TYPE VALUE SETTING

OP	value	rr
4 bits	8 bits	4 bits

INSTRUCTION SET

All LilaKiller instructions are 16-bits (single word) long. This smaller architecture is intended to help introductary programmers with understanding the relationships between human and machine language. The length of the opcode in the processor is four bits, allowing up to 16 different instructions. The instruction set descriptions can be found in **Figure 3.**

INSTRUCTION SYNTAX	EXAMPLE	RESULT
add	add $reg1, $reg2, $reg3	$reg1 + $reg2 -> $reg3
subtract	subtract $reg1, $reg2, $reg3	$reg1 - $reg2 -> $reg3
multiply	multiply $reg1, $reg2, $reg3	$reg1 * $reg2 -> $reg3
divide	divide $reg1, $reg2, $reg3	$reg1 / $reg2 -> $reg3
set	set value, $reg1	value -> $reg1
and	and $reg1, $reg2, $reg3	$reg1 && $reg2 -> $reg3
or	or $reg1, $reg2, $reg3	$reg1 || $reg2 -> $reg3
less than	lessthan $reg1, $reg2, $reg3	if ($reg1 < $reg2); 1 → $reg3 if true, 0 → $reg3 if false.
greater than	greaterthan $reg1, $reg2, $reg3	if ($reg1 > $reg2); 1 → $reg3 if true, 0 → $reg3 if false.
equal to	equalto $reg1, $reg2, $reg3	if ($reg1 == $reg2); 1 → $reg3 if true, 0 → $reg3 if false.
jump	jump $reg1	$reg1 -> PC
store	store $reg1, $reg2	$reg2 -> Mem[$reg1]
load	load $reg1, $reg2	Mem[$reg1] -> $reg2
branch on equal	brancheq $reg1, $zero, $reg2	If ($reg1 == $zero): PC + 2 + 2($reg2) → PC if true, no change → PC if false
jump and link	jumpandlink $reg1	PC + 2 -> $ra $reg1 -> PC

Figure 3: LilaK Instructions

LILAK
ASSEMBLY ARCHITECTURE

LILAKILLER
MICROPROCESSER

THE DATA PATH

To help describe the process of the LilaK pipeline, our team designed a data path diagram that illustrates the five stages of the pipeline.

Below is the five-stage execution pipeline, diagram shown in **Figure 4.**

THE FIVE STAGES

Pipelining processors allows the core to execute an instruction every cycle. As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn, increases the performance.*

The five stages of our pipeline are as follows:

- Fetch
- Decode
- Execute
- Memory
- Writeback

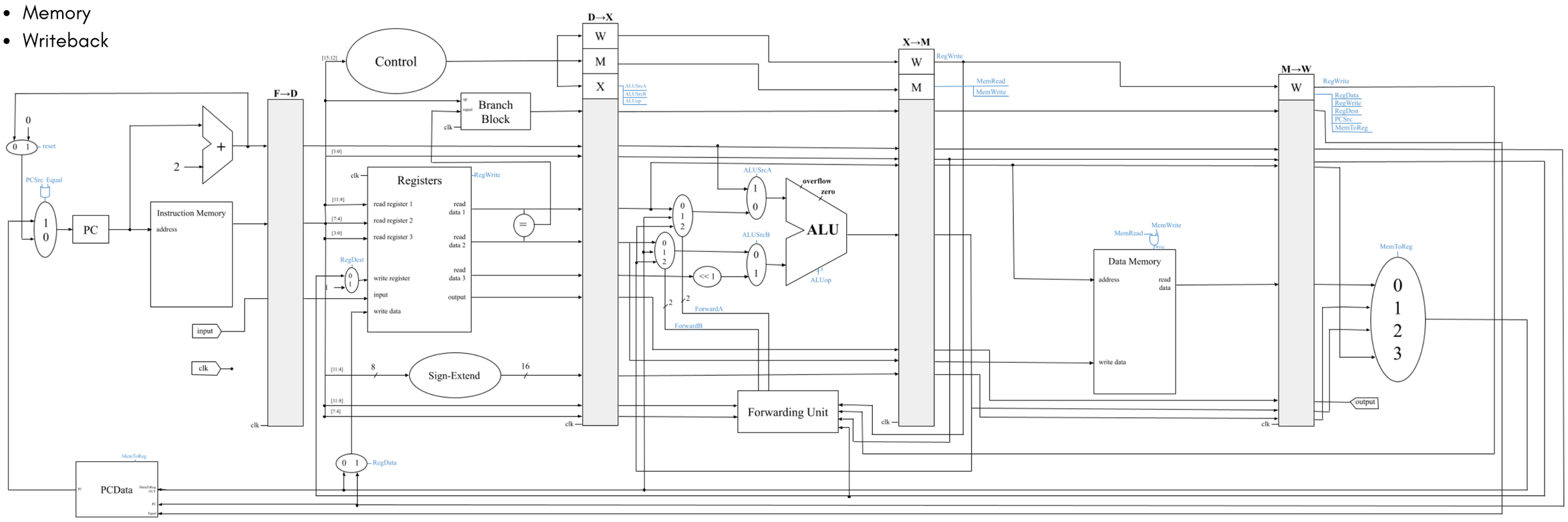


Figure 4: LilaK Data Path

PIPELINE

The LilaK microprocessor utilizes a five-stage execution pipeline, diagram shown in **Figure 4.** Each pipeline stage takes one MasterClock cycle to execute. LilaK does not have any other clock architecture implemented. There is also no latency. Once the pipeline has been completely filled, five instructions will be executing (moving through the pipeline) simultaneously.

While the pipeline is not being stalled, the processor has a throughput of one instruction per MasterClock cycle. The LilaK pipeline, regardless of stalls, is in-order in every step of it's operation: issuance, execution, and completion of instructions are all done in the order of which they entered to the pipeline.

HARDWARE INTEGRATION

A specific integration plan for building the data path was used in order to ensure the efficiency and capabilities of each of the hardware components combined with the rest of the data path.
The data path was built in subsystems, where each stage of the pipeline data path is a subsystem, and the subsystems are connected by stage to stage register files to transfer data between each of the subsystems. These register files were built, tested, and integrated separately from the rest of the system.
Once all of the subsystems, except for the Write Back stage, were correctly implemented and fully tested independent of each other, the subsystems were combined one by one with the stage to stage register files to form the full data path which was tested as one, full unit. The Write Back stage was implemented while combining subsystems to simplify the process. The control unit was added and more tests were run on the fully integrated system including the control signals. This planned integration plan greatly simplified the testing and debugging process and made it much easier to locate errors as the pipeline was being constructed. The sizes of the planned subsystems worked out well and leaving the stage to stage register files to develop, test, and integrate last was the right decision for the group. It also allowed for the integration of additional hardware components that the team realized were necessary during the development process.

REGISTER TRANSFER LANGUAGE

The summary of the LilaK RTL is found in **Figure 5** below.

	ADD, SUBTRACT, MULTIPLY, DIVIDE, AND, OR, GREATER THAN, LESS THAN	SET	JUMP	STORE	LOAD	BRANCH ON EQUAL	JUMP AND LINK
F	INSTRDATA = MEM[PC] PC = PC + 2						
D	A = REG[INSTRDATA[11-8]] B = REG[INSTRDATA[7-4]] C = REG[INSTRDATA[3-0]] CURRENTADDRESS = PC BRANCH = A == B						
X	ALURESULT = A OP B	LOADVALUE = SIGNEXTEND [INSTRDATA[11-4]]				ALURESULT = PC + 2(C)	
M				MEM[A] = B	MEMOUT = MEM[A]		
W	REG[INSTRDATA[3-0]] = ALURESULT	REG[INSTRDATA[3-0]] = LOADVALUE	PC = A			PC = ALURESULT	REG[$RA] = CURRENTADDRESS PC = A

Figure 5: RTL Summary Table

*Source: https://www.sciencedirect.com/topics/computer-science/stage-pipeline#:~:text=The%20pipeline%20allows%20the%20core,in%20turn%20increases%20the%20performance.

LILAK
ASSEMBLY ARCHITECTURE

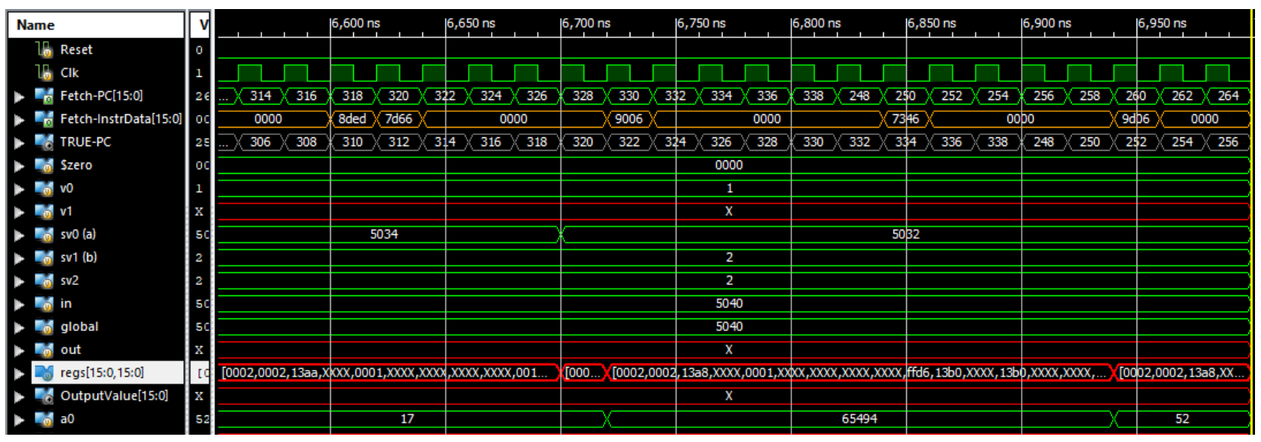
LILAKILLER
MICROPROCESSER

TESTING

For testing our components, we initially created all of our components in Xilinx as schematics and Verilog files and tested their functionality individually with Verilog test fixtures. We tried to test for every possible scenario in these initial tests so that there weren't trivial errors down the road that went undetected. We combined our components into subsystems, mainly the four stages that are in between our register files, so a Fetch stage, Decode stage, Execute stage, and Memory stage. We then tested these by imitating instructions passing through them and testing all of the control bits as well as outputs that result from these instructions. After thoroughly testing each substage, we combined them into smaller versions of our datapath and tested these as well. This is where we discovered and corrected many of the small bugs within our design and refined our architecture as a whole. Once we got the entire datapath implemented, the testing was less in Verilog and more in analyzing the waveforms and deciphering what each instruction should be doing at what stage.

```
RegWriteDataIn = 2; // This is doing the job of the
WriteRegister = 11; // Write Back stage
RegWriteIn = 1;     // for set $v0, 2
#PERIOD; // Now DecodeStage should be outputting the proper control values and inputting them into DtoXRegFile and DtoXRegFile is outputti
RegWriteDataIn = 5; //Doing the job of the
WriteRegister = 12; // Write Back stage for set $v1, 5
#PERIOD;
RegWriteIn = 0;
// Now DtoXRegFile should be outputting all the stored values and inputting them into ExecuteStage & XtoMRegFile @210ns
// ExecuteStage has no clock, so it should do it's computation and output ALUout into XtoMRegFile
// TODO: Test for raout and rbout and Equal (for DtoX)
// TODO: Test for Zero flag and Overflow flag
#PERIOD;
// ALL FINAL VALUES SHOULD BE OUTPUTTED HERE @230ns?
if (rr == 11)
   $display("rr is 11. PASSED. Clock: %d", clockval);
else begin
   $display("rr is not 11. FAILED", raout);
   counter = counter + 1;
end
if (MemRead == 0 && MemWrite == 0 && RegWriteOut == 1 && RegDest == 0 && RegData == 0 && PCSrc == 0 && MemToReg == 0)
   $display("Control bits test for 'set $v0, 2' PASSED");
else begin
   $display("Control bits test for 'set $v0, 2' FAILED");
   counter = counter + 1;
end
if (SEValue == 2)
   $display("Sign Extended Imm test for 'set $v0, 2' PASSED");
else begin
   $display("Sign Extended Imm test for 'set $v0, 2' FAILED");
   counter = counter + 1;
end

// TESTS FOR 'set $v0, 2' COMPLETED
// NOW TESTING 'set $v1, 5'
#PERIOD;
if (rr == 12)
   $display("rr is 12. PASSED. Clock: %d", clockval);
else begin
   $display("rr is not 12. FAILED", raout);
   counter = counter + 1;
end
```



ASSEMBLER/COMPILER FUNCTIONALITY

The assembler reads in each line of the program and translate all pseudoinstructions into their multi-instruction equivalent and then adds nops wherever necessary depending on the instruction and the hazard prevention logic. The assembler uses the specified $a0 register that is reserved for only the assembler. The compiler then translates all the instructions into hexadecimal and binary values. These binary values are written line by line into a separate text file which is automatically loaded into the instruction memory block in the data path. The hexadecimal values are printed to the console for debugging purposes and for the use of the programmer.

The assembler was written in Python because the language has good string manipulation and list comprehension features. It was also a familiar language to the team and easily accessible.

ASSEMBLER/COMPILER CODE SNIPPET

Below is a code-snippet from the assembler and compiler program, written by Dillon Duff. (**Figure 5**)
The three "types" of instructions in LilaK assembly architecture: V-Type instructions, A-Type instructions, and pseudoinstructions, are all translated and converted into their own separate functions before the full translation process. Below is the code for the V-Type and A-Type conversions.

```
def convert_a_type(instr):
    converted = [x if x not in a_type and x not in a_type_dict.keys() else a_type_dict[x] for x in instr]
    if instr[0] == "store":
        converted.append(converted[2])
        converted[3] = "0"
        return '0x' + ''.join(converted)
    elif instr[0] == "jump":
        return '0x' + converted[0] + converted[1] + "00"
    elif instr[0] == "jumpandlink":
        return '0x' + converted[0] + converted[1] + "00"
    elif instr[0] == "load":
        return '0x' + converted[0] + converted[1] + "0" + converted[2]
    return '0x' + ''.join(converted)


def convert_v_type(instr):
    instr = [str(inst).replace(",", "") for inst in instr]
    hex_string = str(hex(int(instr[1]))).replace("x", "")
    hex_string = hex_string[1::]
    if len(hex_string) == 2:
        pass
    elif len(hex_string) == 1:
        hex_string = "0" + hex_string
    else:
        print(f"ERROR: This is not length 1 or 2 in hex... :{hex_string}")
    hex_string = hex_string.upper()
    return f"0x7{hex_string}{a_type_dict[instr[2].replace('$', '')]}"
```

Figure 5: Assembler Conversion Code Snippet

CONCLUSION

Building and developing a new processor was a great experience academically as well as professionally. The team was required to learn and develop new knowledge and skills during the development of the project and this taught everyone how to find necessary information without a lot of additional assistance or provided resources. The team is proud of the final processor and program. With a little bit of extra time, the team would add a new hazard unit that is much more involved than the current solution to hazards in the program and edit the current forwarding unit to work much better and to account for all instructions and possible situations. The current design already accounts for and is prepared to implement a new hazard unit, so a new block would just have to be made and integrated into the final data path. The majority of the problems the team encountered during the completion of the project had to do with weird Xilinx errors which were solved by deleting malfunctioning blocks and replacing them with new ones and version control issues with the git. The schematic files in Xilinx did not work well with the repo and this resulted in the wiping or corruption of files in the project folder that required a lot of time in debugging and recreating.