Single Cycle Datapath Processor using MIPS

Piero Morales
Computer Science University Student
University of Engineering and Technology
Lima, Peru
piero.morales@utec.edu.pe

Angel Motta
Computer Science University Student
University of Engineering and Technology
Lima, Peru
angel.motta@utec.edu.pe

***Abstract*—Two undergraduates implemented a 32 bits pathline based on RISC MIPS for a computer architecture course. This datapath support instructions R-type, I-type and J-type. This included designing the architecture in Verilog, developing test bench modules for the implementation.**

***Index Terms*—Computer architecture, risc, verilog, processor, big endgian, microprocessor without interlocked pipeline stages**

I. Introduction

The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains almost unchanged. The CPU has become the nerve center of any computer, from mobile devices to supercomputers. From the beginning of computer era scientists have tried to improve processor performance not only increasing the number of transistors, but also by improving the instructions that the processor executes. A major change that happened for CPUs is the paradigm from a single core to multi core that increased signifcantly its performance. In this way, Moore's law, that until this moment had traced the future of processors, is discarded. MIPS was originally invented as part of a Standford research project [**?**] and this was the beginig of the new era for CPUs. Due this fact we focus this project to study and understand the detail of this important architecture.

II. Methodology

For this project we use a Hardware Description Language (HDL) to design and simulate our processor and all its components related. The datapath was coded in Verilog. We choose Verilog [**?**] as HDL because is widely used in the industry and it was simulated and tested using test bench modules.

The goal of this project is achieve a better understanding of MIPS single-cycle and implement it with focussing in the basic operations with integers, covering R-type, I-type and J-type instructions for 32 bits MIPS ISA:

A. Datapath

To achieve the goal of supporting all instructions listed before we need to implement the following components:

- Aritmetic Logic Unit (ALU), one of the core componentes of the processor who make the operations of addition, subtraction, comparation between two numbers, logic AND, logic OR, logic NOR.
- Instruction Memory, stores all the instructions to be read and executed according to the address selected.
- PC Counter, a register to hold the address of the current instruction being executed.
- Register File, space that stores 32 registers for MIPS ISA, each one of 32 bits.
- Data Memory, stores the data to support load and stores instructions.
- Multiplexor 2 to 1, determine which of the 2 inputs input select, based on a selector signal i.e. in the selection between the PC Counter, the branch or the jump.
- Adder, execute PC + 4 to link the following instruction, also is used for the offset to cover the branch instruction.

TABLE I
R Type

Instructions		
ADD	Subtraction (SUB)	AND
NOR	OR	Set Less Than (SLT)
Jump Register (JR)		

TABLE II
I Type

Instructions		
Add Immediate (ADDI)	Subtraction Inmediate (SUBI)	AND Inmediate (ANDI)
OR Immediate (ORI)	Set Less Than Immediate (SLTI)	Store Byte (SB)
Store Halfword (SH)	Store Word (SW)	Load Byte (LB)
Load Halfword (LH)	Load Word (LW)	Load Upper Immediate (LUI)
Branch On Equal (BEQ)	Branch On Not Equal (BNEQ)	Branch On Greater than equal zero (BGEZ)

TABLE III
J Type

Instructions		
Jump (J)	Jump and Link (JAL)	—

- Shift Left 2 and 16, to be used to calculate the offset for the branch and load a number up to 32 bits respectively.
- Sign extend, used to extend the most significant bit of the number.
- and the control component for support all the instructions deciding which signal activate depending on the type of instruction and the operation.

The structure of the datapath including all the components:

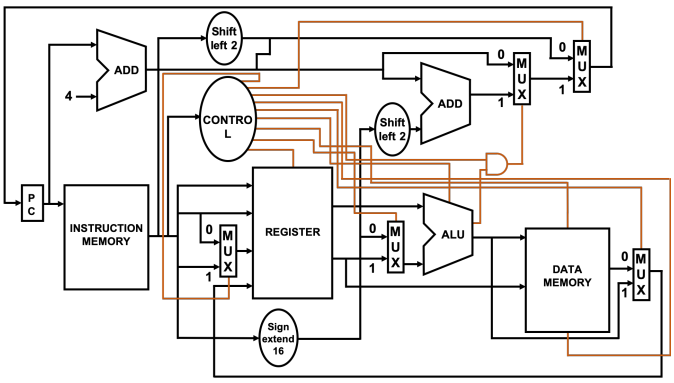

Fig. 1. Datapath.

B. Verilog

The implementation in Verilog is in a single file called `Datapath.v` where is coded the instruction set architecture. Three differents files called `instruction.txt` are used to test and validate the correct functionality of the implementation along with a file called `Test_datapath.v` In total the implementation is composed of the following 5 files:

```
   Datapath.v              (implementation)
Test_datapath.v            (testing module)
instruction.txt      (for arithmetic operations)
instruction.txt   (for load and store operations)
instruction.txt   (for branch and jump operations)
instruction.txt      (test a factorial program)
```

III. Experimental Setup

The implementation of the datapath has the following setup

A. Instructions and opcodes

The opcodes used in the implementation are shown in the Table IV

B. Register File

32 registers, each one of 32 bits.

C. Instruction memory

Size: 256 bytes.

D. Data memory

Size: 256 bytes.

TABLE IV
Opcode MIPS

N	TYPE	OPCODE	OPERATION
1	ADD	000000	add $s0,$t0,$t1
2	SUB	000001	add $s1,$t2,$t3
3	AND	000010	and $s2,$t4,$t5
4	NOR	000011	nor $s3,$t6,$t7
5	OR	000100	or $s4,$t8,$t9
6	SLT	000101	slt $s5,$t0,$t1
7	ADDI	000110	addi $s6,$t2,45
8	SUBI	000111	subi $s7,$t3,37
9	ANDI	001000	andi $s0,$t4,66
10	ORI	001001	ori $s1,$t5,89
11	SLTI	001010	slti $s2,$t6,65535
12	LB	001011	lb $s3,33,($zero)
13	LH	001100	lh $s4,65($zero)
14	LW	001101	lw $s5,48($zero)
15	LUI	001110	lui $s6,346
16	MUL	001111	mul $s2,$s0,$s1
17	SB	010000	sb $s7,24($zero)
18	SH	010001	sh $s0,73,($zero)
19	SW	010010	sw $s1,91,($zero)
20	BEQ	010011	beq $t0,$t1,1
21	BNEQ	010100	bneq $t2,$t3,1
22	BGEZ	010101	bgez $t4,10
23	J	010110	j 12
24	JAL	010111	jal 15
25	JR	011000	jr $ra

E. Testbench

Each test bench is using 5 nanoseconds as a positive clock signal and negative clock signal which give us 10 nanoseconds in total per clock cycle. The implementation has three different test bench for each operation type (arithmetic, load and store, branch and jump) and one final test bench to validate a factorial program.

TABLE V
Testbench 1

Instructions		
ADD	Subtraction (SUB)	AND
NOR	OR	Set Less Than (SLT)
Add Immediate (ADDI)	Subtraction Inmediate (SUBI)	AND Inmediate (ANDI)
OR Immediate (ORI)	Set Less Than Immediate (SLTI)	—

TABLE VI
TESTBENCH 2

Instructions		
Store Byte (SB)	Store Halfword (SH)	Store Word (SW)
Load Byte (LB)	Load Halfword (LH)	Load Word (LW)
Load Upper Inmediate (LUI)		

TABLE VII
TESTBENCH 3

Instructions		
Branch On Equal (BEQ)	Branch On Not Equal (BNEQ)	Branch On Greater than equal zero (BGEZ)
Jump (J)	Jump and Link (JAL)	Jump Register (JR)

To get a more a realistic test of the processor implemented, It will execute a C program that do a factorial operation (figure 2). The factorial program will use a variety of intructions implemented including recursivity technique, it will be the test bench 4.

```
int fact(int n){
        if (n<1)
        return 1;
        else
        return n*factorial(n-1)
        }
variable = factorial(10);
```

Fig. 2. Factorial function - C code.

The factorial program on MIPS is shown in figure 3.

IV. EVALUATION

According to the proposed set of test bench, We calculate the CPU time [**?**] (time processing) for each test bench considering the following equation:

$$Time = PI * CPI * ClockCycleTime$$

Time means execution time measured in seconds per program. PI is Program Instructions (instructions executed for the program), CPI is Clock Cycles per instruction and Clock Cycle time measured in seconds per clock cycle.

We executed the test bench 1, 2, 3 and 4, the results regarding to CPU time is show in table VIII

The results and outputs of Fig. 4, Fig. 5, Fig6, and Fig. 7. are consistent with the results shown in Table VIII, we obtain the expected amount of clock cycles and the time for each test.

```
        addi $a0, $zero, 10
Fact:   subi $sp, $sp, 8
        sw $ra, 4($sp)
        sw $a0,0($sp)
        slti $t0,$a0,1
        beq $t0,$zero,L1
        addi $v0,$zero,1
        addi $sp,$sp,8
        jr $ra
L1:     subi $a0,$a0,1
        jal fact
        lw $a0,0($sp)
        lw $ra,4($sp)
        addi $sp,$sp,8
        mul $v0,$a0,$v0
        jr $ra
```

Fig. 3. Factorial program on MIPS.

TABLE VIII
CPU TIME

	Total instructions	Total of executed instructions (Expected)	Clock Cycles	Clock Cycle Time (ns)	CPU Time (ns)
Test bench 1	11	11	11	10	110
Test bench 2	7	7	7	10	70
Test bench 3	20	100	100	10	1000
Test bench 4	18	131	131	10	1310

V. CONCLUSION

- A team of 2 undergraduates designed and implemented and tested a 32-bits MIPS processor. The implementation was completed as part of an academic semester-long Computer Architecture course.
- This implementation of single-cycle datapath is a close replica of the original in the early days of RISC architecture. Nowadays this approach show limitations of performance due to the execution of 1 instruction per cycle and this implementation is not considering pipeline technique to improve the performance.
- The implementation successfully passed all tests bench including a factorial program which used many components of the architecture.

VI. COMMENTS

- One of the most challenging actitivities of this project was translate a factorial program from C to machine language, because it required test our whole implementation. This included grasp the real work done behind the scenes for some MIPS instructions like jr or jal.

```
greendev:Program_test1 angelinux$ ./a.out
Clock: 1 Reg 1:         47 Reg 2:          5 Result:         52 PC:   4 Opcode: 000000 Type: R-ADD
Clock: 1 Reg 1:         67 Reg 2:         31 Result:         36 PC:   8 Opcode: 000001 Type: R-SUB
Clock: 1 Reg 1:         36 Reg 2:         80 Result:          0 PC:  12 Opcode: 000010 Type: R-AND
Clock: 1 Reg 1:         29 Reg 2:         63 Result: 4294967232 PC:  16 Opcode: 000011 Type: R-NOR
Clock: 1 Reg 1:         11 Reg 2:         85 Result:         95 PC:  20 Opcode: 000100 Type: R-OR
Clock: 1 Reg 1:         47 Reg 2:          5 Result:          0 PC:  24 Opcode: 000101 Type: R-SLT
Clock: 1 Reg 1:         67 Reg 2:         45 Result:        112 PC:  28 Opcode: 000110 Type: I-ADDI
Clock: 1 Reg 1:         31 Reg 2:         13 Result:         18 PC:  32 Opcode: 000111 Type: I-SUBI
Clock: 1 Reg 1:         36 Reg 2:         66 Result:          0 PC:  36 Opcode: 001000 Type: I-ANDI
Clock: 1 Reg 1:         80 Reg 2:         89 Result:         89 PC:  40 Opcode: 001001 Type: I-ORI
Clock: 1 Reg 1:         29 Reg 2:      65535 Result:          1 PC:  44 Opcode: 001010 Type: I-SLTI
```

Fig. 4. Execution results for test bench 1.

```
greendev:Program_test2 angelinux$ ./a.out
Clock: 1 Result:          65 PC:   4 Opcode: 001011 Type: I-LB
Clock: 1 Result:        6209 PC:   8 Opcode: 001100 Type: I-LH
Clock: 1 Result:   538460420 PC:  12 Opcode: 001101 Type: I-LW
Clock: 1 Result:    22675456 PC:  16 Opcode: 001110 Type: I-LUI
Clock: 1 Result:    22675456 PC:  20 Opcode: 010000 Type: I-SB
Clock: 1 Result:    22675456 PC:  24 Opcode: 010001 Type: I-SH
Clock: 1 Result:    22675456 PC:  28 Opcode: 010010 Type: I-SW
```

Fig. 5. Execution results for test bench 2.

- To find and fix bugs we had to have clear outputs (monitor statement) in our test bench and also have a clear understanding of the value (actual data) of the registers.
- The choice of using icarus verilog was natural to us since we prefered the flexibility of a console environment in a Linux system.

REFERENCES

[1] M. Horowitz, et. al., "MIPS-X: 20-MIPS peak, 32 bits microprocessor," IEEEJournal of Solid-State Circuits, vol. 22, no. 5, pp. 790-799, Oct. 1987.

[2] IEEE Standard for Verilog Hardware Description Language. IEEE Standard 1364-2005 (Revision of IEEE Standard 1364-2001). http://dx.doi.org/10.1109/IEEESTD.2006.99495, 2006. Last access 26 November 2018.

[3] Patterson, D., Hennessy, J. and Alexander, P. (2012). Computer organization and design. 4th ed. Waltham, Mass: Morgan Kaufmann, pp.35.


```
Clock: 1 Result:          0 PC:   4 Opcode: 000000 Type: R-ADD
Clock: 1 Result:   69212225 PC:   8 Opcode: 001101 Type: I-LW
Clock: 1 Result:          0 PC:  12 Opcode: 000101 Type: R-SLT
Clock: 1 Result:          0 PC:  16 Opcode: 010011 Type: I-BEQ
Clock: 1 Result:  138424450 PC:  20 Opcode: 000000 Type: R-ADD
Clock: 1 Result:  138424450 PC:  28 Opcode: 010110 Type: J-J
Clock: 1 Result:  138424450 PC:  32 Opcode: 010010 Type: I-SW
Clock: 1 Result:         93 PC:  36 Opcode: 000110 Type: I-ADDI
Clock: 1 Result:          1 PC:  40 Opcode: 000110 Type: I-ADDI
Clock: 1 Result:          1 PC:  44 Opcode: 010100 Type: I-BNEQ
Clock: 1 Result:          1 PC:   4 Opcode: 010110 Type: J-J
Clock: 1 Result:   69212225 PC:   8 Opcode: 001101 Type: I-LW
Clock: 1 Result:          0 PC:  12 Opcode: 000101 Type: R-SLT
Clock: 1 Result:          0 PC:  16 Opcode: 010011 Type: I-BEQ
Clock: 1 Result:  138424450 PC:  20 Opcode: 000000 Type: R-ADD
Clock: 1 Result:  138424450 PC:  28 Opcode: 010110 Type: J-J
```

Fig. 6. Execution results for test bench 3.

```
Clock: 1 Result:          9 PC:  48 Opcode: 001101 Type: I-LW
Clock: 1 Result:         44 PC:  52 Opcode: 001101 Type: I-LW
Clock: 1 Result:        111 PC:  56 Opcode: 000110 Type: I-ADDI
Clock: 1 Result:     362880 PC:  60 Opcode: 001111 Type: R-MUL
Clock: 1 Result:     362880 PC:  44 Opcode: 011000 Type: R-JR
Clock: 1 Result:         10 PC:  48 Opcode: 001101 Type: I-LW
Clock: 1 Result:         44 PC:  52 Opcode: 001101 Type: I-LW
Clock: 1 Result:        119 PC:  56 Opcode: 000110 Type: I-ADDI
Clock: 1 Result:    3628800 PC:  60 Opcode: 001111 Type: R-MUL
greendev:Test_factorial angelinux$
```

Fig. 7. Execution results for test bench 4 (Factorial of 10).