# Implementation of a MIPS processor in VHDL

This laboratory work describes the design of a simplified MIPS processor and some guidelines for its implementation in VHDL. The outcome will be an implementation of the simplified MIPS processor, which will be tested through simulation.

## 1. MIPS instruction set architecture

MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC (Reduced Instruction Set Computer) architecture. This architecture defines 32 general purpose registers. The first register $r0 always contains the value zero. MIPS has fixed width instructions (32 bit). There are 3 instruction types: I-type (immediate), R-type (register), J-type (jump). For the scope of this laboratory work, only I-type and R-type will be described and used. Fig. 1 shows the format of I-type and R-type instructions.

| opcode | rs | rt | Address/immediate |
|---|---|---|---|
| 6 | 5 | 5 | 16 |

**I-Type instruction**

| opcode | rs | rt | rd | Shift | Function |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

**R-Type instruction**

Figure 1. I-type and R-type instruction formats

MIPS is a load/store architecture, meaning that all operations are performed on values found in local registers. The main memory is only accessed through load (copy value from memory to local register) and store (copy value from local register to memory) instructions.

The fields in the MIPS instructions are the following:

- OPCODE – 6 bit operation code
- RS – 5 bit specifier for source register
- RT – 5 bit specifier for target register
- RD – 5 bit specifier for destination register
- Address/immediate – 16 bit signed immediate used for logical and arithmetic operands, load/store address offsets
- Shift – 5 bit shift amount
- Function – 6 bit code used to specify functions

## 2. Instruction execution

Each instruction is divided into a series of steps:

- Instruction fetch: fetch the instruction from the memory and compute the address of the next instruction.
- Instruction decode: registers indicated by rs and rd are read.
- Execution: the instruction is known, so the function is executed (memory address computation, arithmetic-logical operation).
- Memory access: the memory is accessed based on the address computed before, or the result is written in the destination register.
- Write back: the load operation is completed by writing the value from the memory in the register.

Each step of instruction execution is performed in a clock cycle. The datapaths are presented below.

**Fetch datapath**

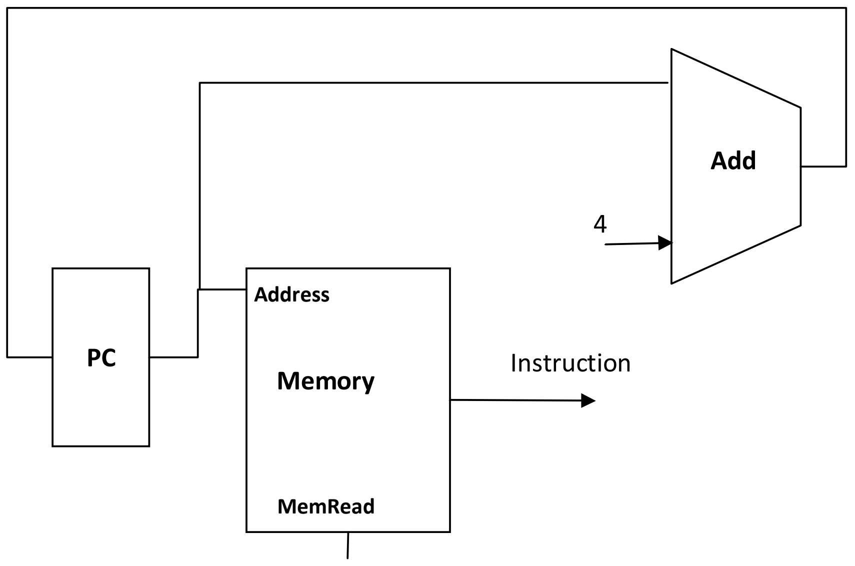

Figure 2. Instruction fetch datapath

**R-type datapath**

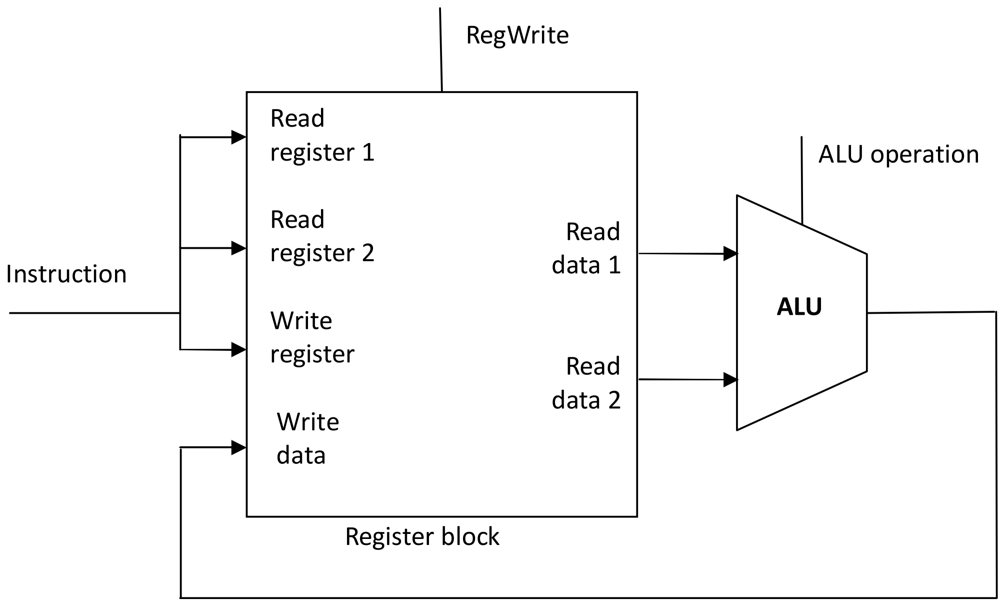

Figure 3. R-type datapath

**Load/store datapath**

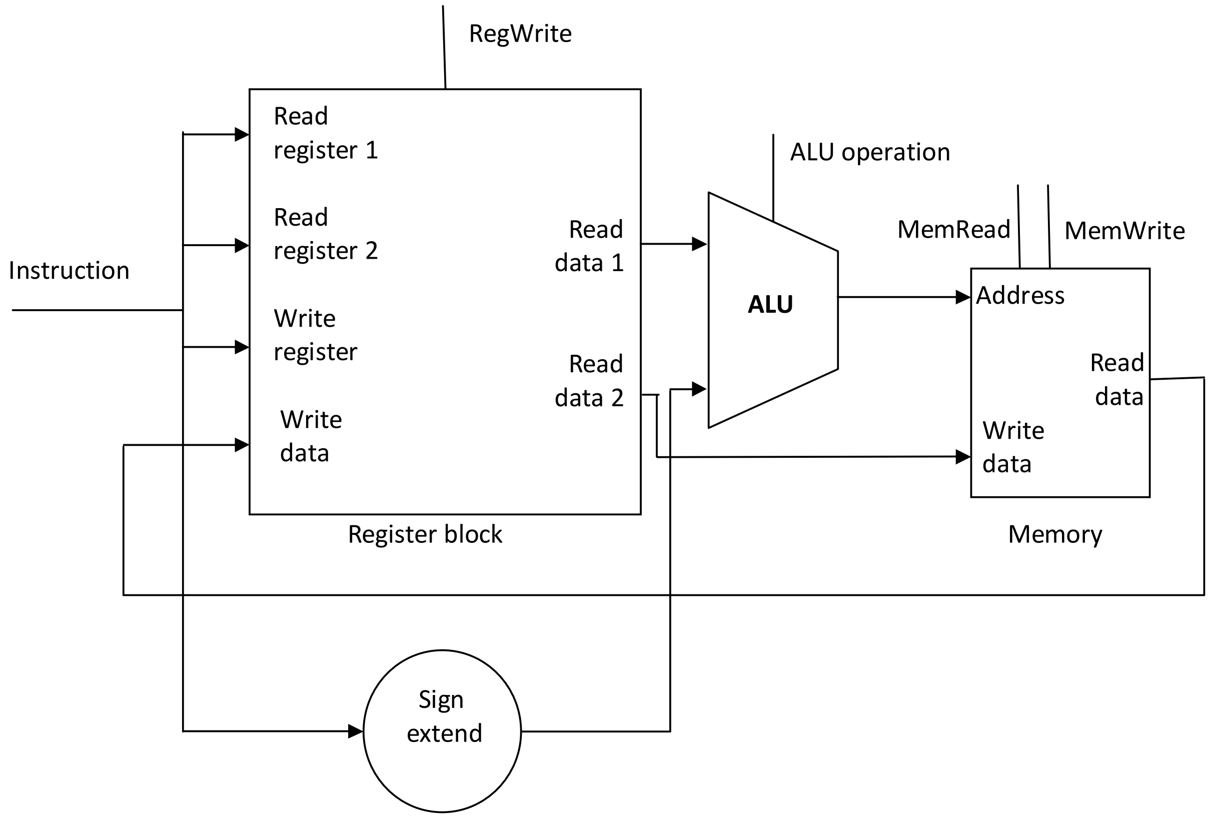

Figure 4. Load/Store datapath

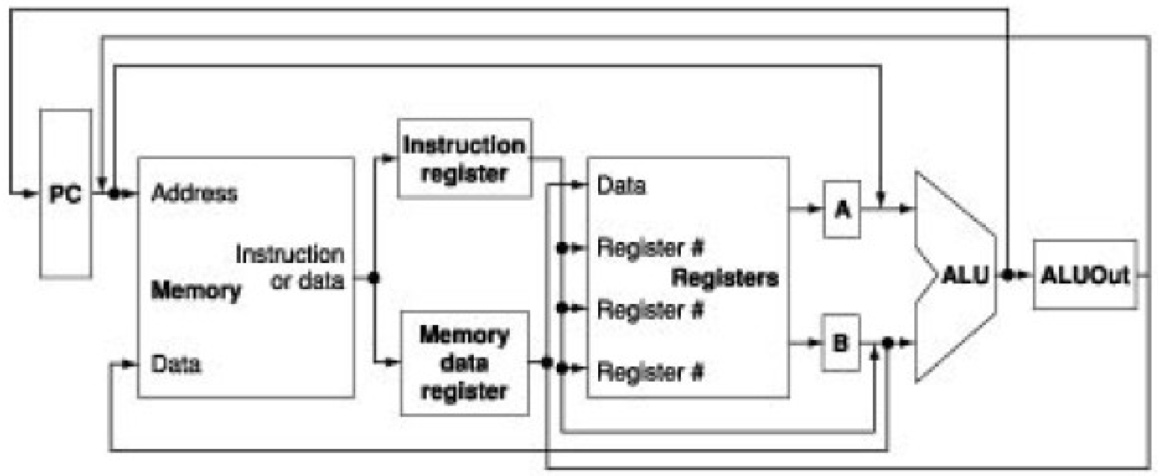

Figure 5. MIPS multicycle datapath

## 3. Assignments

### Design a phase generator

The phase generator will generate the sequence of execution steps as states, based on the clock signal. There will be a distinct state for each step. Design the phase generator and implement it in VHDL.

### Design the control unit

The control unit will generate control signals for each component in the design, based on the current state (given by the phase generator) and on the instruction code. The control signals are the following:

- MemRead: if 1, read from memory;
- MemWrite: if 1, write to memory;
- RegDst: if 0, the register file destination number for the Write register comes from the rt field; if 1, it comes from rd field;
- RegWrite: if 1, the general-purpose register selected by the Write register number is written with the value of the Write data input;
- AluSrcA: if 0, the operand is PC; if 1, the operand is A register;
- AluSrcB: if 0, the operand is 4; if 1, the operand is B register;
- MemtoReg: if 0, The value fed to the register file Write data input comes from ALUOut; if 1, it comes from Memory data register (MDR);
- IRWrite: if 1, write instruction in IR;
- PCWrite: if 1, write the PC;
- ALU control (see ALU control unit design)

| Step | R-Type actions | Memory reference actions |
|---|---|---|
| Fetch | IR = Memory[PC]<br>PC = PC + 4 | |
| Decode | A = Reg[IR[25-21]]<br>B = Reg[IR[20-16]] | |
| Execution | ALUOut = A op B | ALUOut = A + sign-extend (IR[15-0]) |
| Memory | Reg[IR[15-11]] = ALUOut | Load: MDR = Memory[ALUOut]<br>or<br>Store: Memory[ALUOut] = B |
| Write back | | Load: Reg[IR[20-16]] = MDR |

Design the state machine for the control unit, and then implement it in VHDL.

### Design an ALU control unit

The table for the ALU control is the following:

| Instruction | opcode | function | ALU action | ALUop |
|---|---|---|---|---|
| Load | 100011 | - | add | 00 |
| Store | 101011 | - | add | 00 |
| R-Type/add | 000000 | 100000 | add | 00 |
| R-Type/sub | 000000 | 100010 | sub | 01 |
| R-Type/and | 000000 | 100100 | and | 10 |

|  |  |  |  |  |
|---|---|---|---|---|
| R-Type/or | 000000 | 100101 | or | 11 |

Design the state machine for the ALU control unit, and then implement it in VHDL. It can be part of the main control unit.

**Implement the MIPS**

Given previous design, implement the MIPS in VHDL. There are some already implemented components: PC, IR, ALU, Register block (from [1]). At first, do not use a real memory module; just generate some instructions and data to test your implementation. Simulate the VHDL code.

**Constants**

```
PACKAGE ProcMem_definitions IS
-- globals
CONSTANT width : NATURAL := 32;
-- definitions for regfile
CONSTANT regfile_depth : positive := 32; -- register file depth = 2**adrsize
CONSTANT regfile_adrsize : positive := 5; -- address vector size = log2(depth)
END ProcMem_definitions;
```

**The ALU**

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
-- use package
USE work.procmem_definitions.ALL;
ENTITY alu IS
PORT (
a, b : IN STD_ULOGIC_VECTOR(width-1 DOWNTO 0);
opcode : IN STD_ULOGIC_VECTOR(1 DOWNTO 0);
result : OUT STD_ULOGIC_VECTOR(width-1 DOWNTO 0);
zero : OUT STD_ULOGIC);
END alu;

ARCHITECTURE behave OF alu IS
BEGIN
PROCESS(a, b, opcode)
-- declaration of variables
VARIABLE a_uns : UNSIGNED(width-1 DOWNTO 0);
VARIABLE b_uns : UNSIGNED(width-1 DOWNTO 0);
VARIABLE r_uns : UNSIGNED(width-1 DOWNTO 0);
VARIABLE z_uns : UNSIGNED(0 DOWNTO 0);
BEGIN
-- initialize values
a_uns := UNSIGNED(a);
```

```
b_uns := UNSIGNED(b);
r_uns := (OTHERS => '0');
z_uns(0) := '0';
-- select desired operation
CASE opcode IS
-- add
WHEN "00" =>
r_uns := a_uns + b_uns;
-- sub
WHEN "01" =>
r_uns := a_uns - b_uns;
-- and
WHEN "10" =>
r_uns := a_uns AND b_uns;
-- or
WHEN "11" =>
r_uns := a_uns OR b_uns;
-- others
WHEN OTHERS => r_uns := (OTHERS => 'X');
END CASE;
-- set zero bit if result equals zero
IF TO_INTEGER(r_uns) = 0 THEN
z_uns(0) := '1';
ELSE
z_uns(0) := '0';
END IF;
-- assign variables to output signals
result <= STD_ULOGIC_VECTOR(r_uns);
zero <= z_uns(0);
END PROCESS;
END behave;
```

**The PC**

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
-- use package
USE work.procmem_definitions.ALL;
ENTITY pc IS
PORT (
clk : IN STD_ULOGIC;
rst_n : IN STD_ULOGIC;
pc_in : IN STD_ULOGIC_VECTOR(width-1 DOWNTO 0);
PC_en : IN STD_ULOGIC;
pc_out : OUT STD_ULOGIC_VECTOR(width-1 DOWNTO 0) );
END pc;
```

```vhdl
ARCHITECTURE behave OF pc IS
BEGIN
proc_pc : PROCESS(clk, rst_n)
VARIABLE pc_temp : STD_ULOGIC_VECTOR(width-1 DOWNTO 0);
BEGIN
IF rst_n = '0' THEN
pc_temp := (OTHERS => '0');
ELSIF RISING_EDGE(clk) THEN
IF PC_en = '1' THEN
pc_temp := pc_in;
END IF;
END IF;
pc_out <= pc_temp;
END PROCESS;
END behave;
```

**The IR**

```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
-- use package
USE work.procmem_definitions.ALL;
ENTITY instreg IS
PORT (
clk : IN STD_ULOGIC;
rst_n : IN STD_ULOGIC;
memdata : IN STD_ULOGIC_VECTOR(width-1 DOWNTO 0);
IRWrite : IN STD_ULOGIC;
instr_31_26 : OUT STD_ULOGIC_VECTOR(5 DOWNTO 0);
instr_25_21 : OUT STD_ULOGIC_VECTOR(4 DOWNTO 0);
instr_20_16 : OUT STD_ULOGIC_VECTOR(4 DOWNTO 0);
instr_15_0 : OUT STD_ULOGIC_VECTOR(15 DOWNTO 0) );
END instreg;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
-- use package
USE work.procmem_definitions.ALL;
ARCHITECTURE behave OF instreg IS
BEGIN
proc_instreg : PROCESS(clk, rst_n)
BEGIN
IF rst_n = '0' THEN
instr_31_26 <= (OTHERS => '0');
```

```
instr_25_21 <= (OTHERS => '0');
instr_20_16 <= (OTHERS => '0');
instr_15_0 <= (OTHERS => '0');
ELSIF RISING_EDGE(clk) THEN
-- write the output of the memory into the instruction register
IF(IRWrite = '1') THEN
instr_31_26 <= memdata(31 DOWNTO 26);
instr_25_21 <= memdata(25 DOWNTO 21);
instr_20_16 <= memdata(20 DOWNTO 16);
instr_15_0 <= memdata(15 DOWNTO 0);
END IF;
END IF;
END PROCESS;
END behave;
```

**The Register block**

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
-- use package
USE work.procmem_definitions.ALL;
ENTITY regfile IS
PORT (clk,rst_n : IN std_ulogic;
wen : IN std_ulogic; -- write control
writeport : IN std_ulogic_vector(width-1 DOWNTO 0); -- register input
adrwport : IN std_ulogic_vector(regfile_adrsize-1 DOWNTO 0);-- address write
adrport0 : IN std_ulogic_vector(regfile_adrsize-1 DOWNTO 0);-- address port 0
adrport1 : IN std_ulogic_vector(regfile_adrsize-1 DOWNTO 0);-- address port 1
readport0 : OUT std_ulogic_vector(width-1 DOWNTO 0); -- output port 0
readport1 : OUT std_ulogic_vector(width-1 DOWNTO 0) -- output port 1
);
END regfile;

ARCHITECTURE behave OF regfile IS
SUBTYPE WordT IS std_ulogic_vector(width-1 DOWNTO 0); -- reg word TYPE
TYPE StorageT IS ARRAY(0 TO regfile_depth-1) OF WordT; -- reg array TYPE
SIGNAL registerfile : StorageT; -- reg file contents
BEGIN
-- perform write operation
PROCESS(rst_n, clk)
BEGIN
IF rst_n = '0' THEN
FOR i IN 0 TO regfile_depth-1 LOOP
registerfile(i) <= (OTHERS => '0');
END LOOP;
ELSIF rising_edge(clk) THEN
IF wen = '1' THEN
```

```
registerfile(to_integer(unsigned(adrwport))) <= writeport;
END IF;
END IF;
END PROCESS;
-- perform reading ports
readport0 <= registerfile(to_integer(unsigned(adrport0)));
readport1 <= registerfile(to_integer(unsigned(adrport1)));
END behave;
```

## Bibliography

[1] M. Linder, M. Schmid, “Processor Implementation in VHDL”