Instruction Set Architecture Design and Implementation

Learning Outcomes

At the end of this lab, you should be able to:

1. understand and modify a simple ISA and its implementation;
2. write a simple assembler

Contents

1 Resources

- Video: https://youtu.be/CzyXb_T-xgU.

- Source Codes: https://git.io/JU3al

- Online VHDL Tool: https://www.edaplayground.com/home

---

2 Discussion

The **processor(CPU)** is composed of the **datapath** and **control**. In the previous labs, you learned that combinational and sequential circuit elements are used as building blocks to create the functional components of the datapath and control. Examples of these functional elements include the **ALU, Register File, Program Counter, and Memory**. You also learned that a **clock** drives the execution and control is implemented using a **finite state machine** for **fetch-decode-execute cycle**. One question that we can answer next is: *How do we program the CPU?*

2.1 Instruction Set Architecture

Instruction Set Architecture (ISA) is an **abstraction** between the hardware and the lowest-level software. It includes anything programmers need to know to make a binary machine language program work correctly. Typically it documents the set of instructions that can be performed by the processor, number and name of available registers, memory addressing modes, I/O, interrupt processing, etc.

ISA allows computer designers to talk about functions indepedently from the hardware that performs them. This abstact interface enables many implementations (aka **microarchitectures**) of varying costs and performance to run identical software.

Examples of ISA include the **IA-32** and **x86-64** which are commonly used in desktop and laptops. Intel implements these ISA in their Intel Core i5 product as **8th Generation aka as Kaby Lake Refresh**. AMD also implements these ISA in the Ryzen 5000 as **4th Gen aka Zen 3**. There are other implementations(aka generations) that vary in their performance characteristics.

For mobile devices, a popular ISA is the **ARMv8 A64**. MediaTek uses ARM's **Cortex-A73** and **Cortex-A53** implementations in their Octa-core Helio P70. Qualcomm also uses the same implementations in their Kryo 240 processor for Snapdragon SoC.

2.2 Application Binary Interface

Application Binary Interface (ABI) is a combination of the basic instruction set and the operating system interface provided for application programmers.

For general-purpose use such a desktops and laptops, programming a processor using only the basic instruction set is inefficient. Thus, operating systems perform an important role in the management and efficient use of hardware resources in addition to making it easier for users to use a computer.

ABI describes function-calling conventions, parameter passing, sizes of C data types, executable file formats (ELF, PE). Examples are the IA-32 and **x86-64 System V ABI** which is used in Linux and other Unix-type operating systems. In Windows, it uses its own **x64 ABI**.

2.3 ISA Taxonomy

We can categorize ISAs based on where operands in instructions are stored. **Stack-based** ISAs uses a stack(LIFO) where operations are performed on the operands on the top of the stack. In **accumulatr-based** ISAs, one register is designated as accumulator and its use in operations is implied. Modern ISAs are **general purpose** where operands are explicitly named in the instruction. Operations can be register-to-register, register-to-memory, or memory-to-register.

2.4 Considerations in ISA Design

- *Types/Class of instructions(Operations in the instruction set)* - arithmetic/logic, data movement, branching/control flow, I/O, etc.

- *Types and sizes of operands* - 8, 16, 32, 64, 128, floating point

- *Addressing modes* - register, direct, indirect, immediate, etc.

- *Addressing memory* - byte-addressable, word-addressable

- *Encoding and Instruction Formats* - opcode field, addresses field, mode field

- *Compiler-related issues* - optimization features

2.5 Example Instruction Formats

Figure 1 and Figure 2 are the instruction formats for x86-64 and ARMv8, taken from their documentation manuals. The x86-64 format is more complex than that of the ARMv8 with variable widths in terms of number of bits for the opcode.
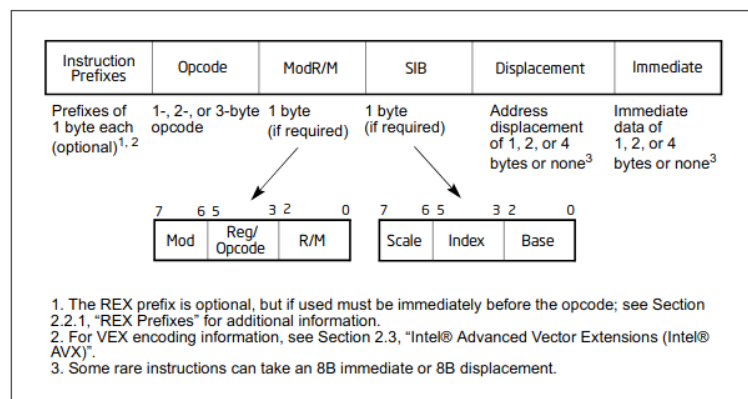

Figure 1: x86-64 Instruction Format (CISC).

Figure 2: ARMv8 Instruction Format (RISC).

2.6 The Optimal Machine Architecture (TOMA) ISA

Let us look at the design of a simple ISA called TOMA.

2.6.1 Features

- Four 8-bit registers named $s0, $s1, $s2, $s3 when used in assembly code

- Instruction memory (IM) is 8 bytes(8x8), address line is 3 bits

- Three-bit Program Counter (PC)

- Single-cycle - completes instruction execution in one clock cycle

- Supports the following instructions: and, add, sub, addi

- No data memory, thus has no load and store instructions

- No control transfer instructions

2.6.2 Instruction Format

The size of an instruction in TOMA is 8 bits divided into the configuration shown in Figure 3.

- Bits 7,6 - opcode (op)
- Bits 5,4 - source register 1 (rs)
- Bits 3,2 - source register 2 (rt)
- Bits 1,0 - destination register (rd)/immediate

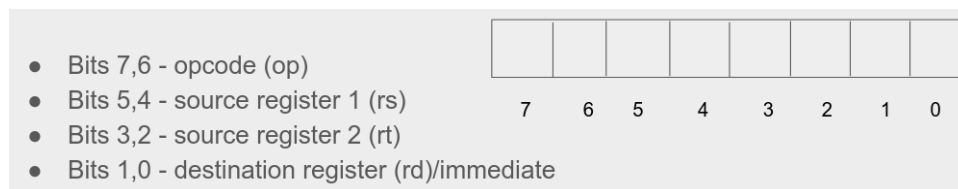|							
7	6	5	4	3	2	1	0

Figure 3: TOMA instruction format.

2.6.3 Supported Instructions

Listing 1: Supported Instructions in TOMA.

```
and : rd <= rs AND rt          (op=00)
add : rd <= rs + rt            (op=01)
sub : rd <= rs -  rt           (op=10)
addi : rs <= rt + immediate    (op=11)
```

2.6.4 Example assembly code and machine code

Listing 2: Example assembly code and machine code.

```
addi $s0, $s0, 2        ; 11000010b, 0xC2
addi $s1, $s1, 1        ; 11010101b, 0xD5
addi $s2, $s2, 3        ; 11101011b, 0xEB
add $s3, $s0, $s1       ; 01000111b, 0x47
sub $s0, $s2, $s3       ; 10101100b, 0xAC
```

The syntax for the assembly language isn shown in Listing 3.

Listing 3: Example assembly code and machine code.

```
<instruction> <dst>, <src1>, <src2/imm>
```

2.7 REDHORSE 500: An implementation of TOMA

Given the TOMA ISA, let us implement the ISA and call it REDHORSE 500. We will use VHDL for the implementation.

2.7.1 Processor

The interface to the processor is shown in Listing 15 which shows that it has one input and two outputs. Figure 10 shows the complete wiring of the functional components.

Listing 4: Interface to the processor.

```
ENTITY Processor IS
    PORT
    (
        clk :  IN  STD_LOGIC;
        current_instruction :  OUT  STD_LOGIC_VECTOR(2 DOWNTO 0);
        value :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END Processor;
```

Figure 4: REDHORSE 500.

2.7.2 Program Counter

Listing 5: Interface to the Program Counter.

```vhdl
entity PC is
  port(
    clk          : in std_logic;
    current_instr : in std_logic_vector(2 downto 0);   -- current instruction
    next_instr    : out std_logic_vector(2 downto 0)   -- next instruction
  );
end PC;
```
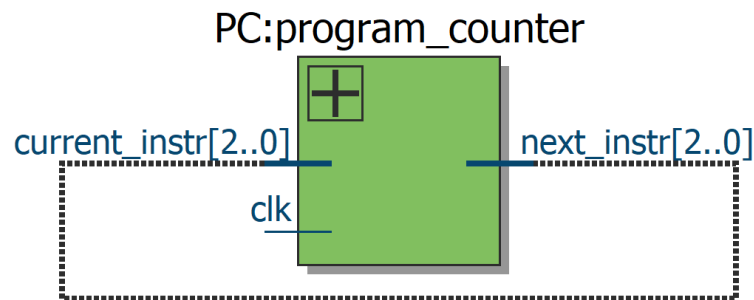

Figure 5: Program Counter.

2.7.3 Instruction Memory

Listing 6: Interface to the Instruction Memory.

```vhdl
entity Instruction is
  port(
    instr_addr : in std_logic_vector(2 downto 0);  -- instruction address

    op       : out std_logic_vector(1 downto 0); -- operation code
    rs_addr  : out std_logic_vector(1 downto 0); -- source register 1 addr
    rt_addr  : out std_logic_vector(1 downto 0); -- source register 2 addr
    rd_addr  : out std_logic_vector(1 downto 0)  -- dest register addr
  );
end Instruction;
```
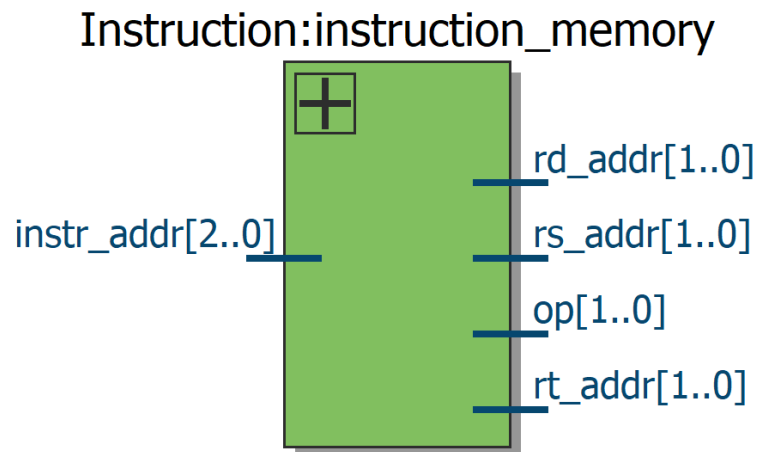
Instruction:instruction_memory

Figure 6: Instruction Memory.

Listing 7: Hard-coded instruction in the instruction memory.

```
constant instr : instruction_set := (
  "11000010",  -- addi $s0, $s0, 2
  "11010101",  -- addi $s1, $s1, 1
  "11101011",  -- addi $s2, $s2, 3
  "01000111",  -- add  $s3, $s0, $s1
  "10101100",  -- sub  $s0, $s2, $s3
  "00000000",
  "00000000",
  "00000000"
);
```

2.7.4   Register File

Listing 8: Interface to the Register File.

```vhdl
entity Registers is
  port(
    clk     : in std_logic;

    rs_addr : in std_logic_vector(1 downto 0);      -- source register 1 address
    rt_addr : in std_logic_vector(1 downto 0);      -- source register 2 address
    rd_addr : in std_logic_vector(1 downto 0);      -- destion register address
    wr_data : in std_logic_vector(7 downto 0);      -- write data to dest register

    rs      : out std_logic_vector(7 downto 0);     -- source register 1 value
    rt      : out std_logic_vector(7 downto 0)      -- source register 2 value
  );
end Registers;
```
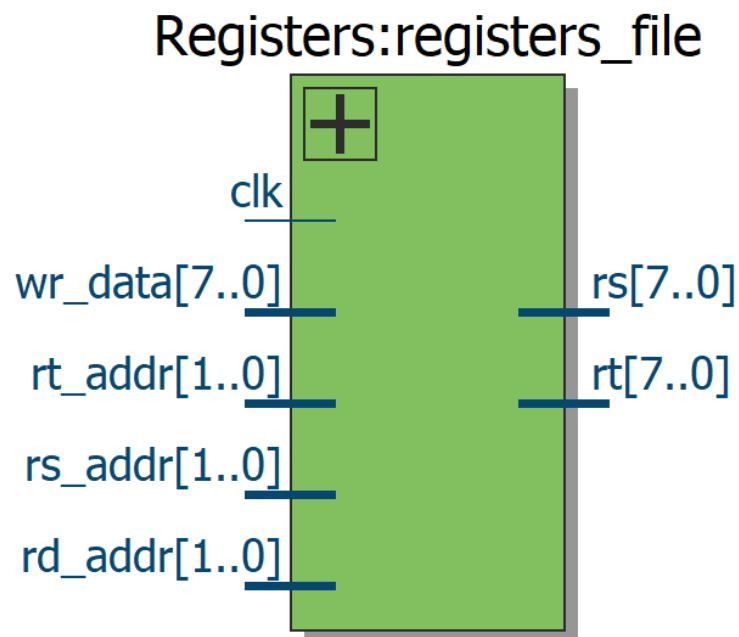

Figure 7: Register File.

Listing 9: Hard-coded initial values for the registers.

```
    signal reg: registerFile := (
     "00000001",
     "00000010",
     "00000011",
     "00000100"
);
```

2.7.5 ALU

Listing 10: Interface to the ALU.

```
entity ALU is
  port(
    op  : in std_logic_vector(1 downto 0);   -- operation code

    rs  : in std_logic_vector(7 downto 0);   -- source register 1
    rt  : in std_logic_vector(7 downto 0);   -- source register 2
    rd  : out std_logic_vector(7 downto 0)   -- destination register
  );
end ALU;
```
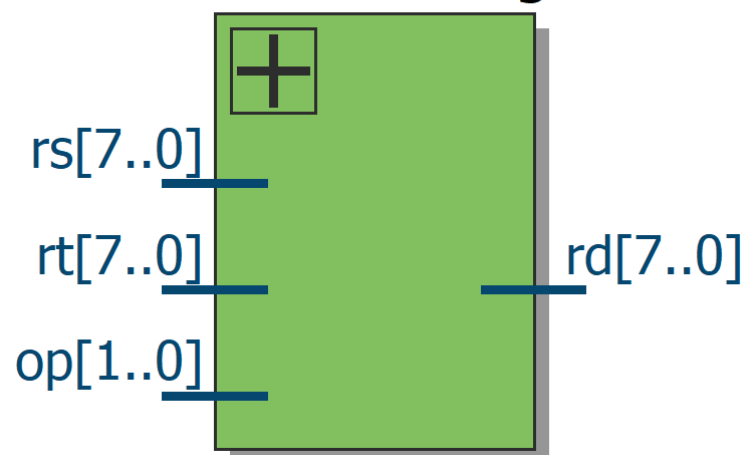

Figure 8: ALU.

2.7.6 Control Unit

Listing 11: Interface to the Control Unit.

```vhdl
entity Control is
  port(
    instr   : in std_logic_vector(1 downto 0);   -- instruction

    alu_op  : out std_logic_vector(1 downto 0);  -- operation code of AlU
    alu_src : out std_logic;                      -- ALU select ADDi
    reg_dst : out std_logic                       -- select destination address
    register
  );
end Control;
```
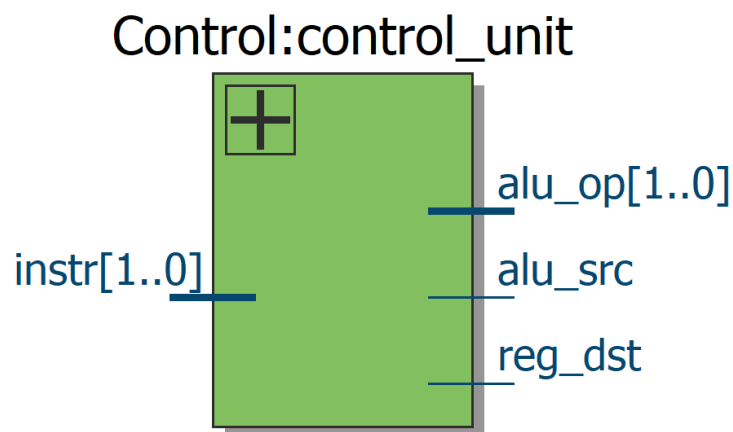

Figure 9: Control Unit.

2.7.7 Mux0

Listing 12: Interface to Mux0.

```vhdl
entity mux0 is
  port(
    sel : in std_logic;                          -- select destination address
    a   : in std_logic_vector(1 downto 0);       -- source register address
    b   : in std_logic_vector(1 downto 0);       -- default destination address
    y   : out std_logic_vector(1 downto 0)       -- destination address
  );
end mux0;
```

2.7.8 Mux1

Listing 13: Interface to Mux1.

```vhdl
entity mux1 is
  port(
    sel : in std_logic;                      -- select data
    a   : in std_logic_vector(7 downto 0);   -- default data
    b   : in std_logic_vector(7 downto 0);   -- data from instruction
    y   : out std_logic_vector(7 downto 0)   -- data out
  );
end mux1;
```

2.7.9 Sign Extend

Listing 14: Interface to Sign Extend

```vhdl
entity sign_extend is
  port(
    data_in  : in std_logic_vector(1 downto 0);
    data_out : out std_logic_vector(7 downto 0)
  );
end sign_extend;
```

3 Summary

In this lab, you learned some of the sequential elements that are useful in the design of a processor as well as the importance of clocks. We also showed the design and implementation of a simple traffic light system using finite state machines since a simple truth table is not enough to characterize a sequential system.

You should now be able to tell whether a functional component of a datapath and control is composed of a combinational or sequential element.

4 Learning Activities

Download the source codes for this lab then try experimenting by adding more test cases in the testbenches. Submit a PDF document that shows screenshots of your modifications and runs.

5 Self-Assessment Questions

1. What is the main purpose of clocks in sequential circuits?

2. What is the difference between a clocked latch and a flip-flop?

3. Why can't a multiplexer be used in RAM?

4. Why is SRAM more expensive than DRAM?

5. If my CPU is clocked at 800 MHz, what is the period?

6   Deliverable

Your final deliverable for this lab is implement the RAM in Figure **??**. Submit the VHDL code including a testbench as well as images of the waveforms. NOTE: Enable lines should be connected to the output of the decoder and the rightmost Din in the figure should be Din[0].

7   Further Reading

- https://www.doulos.com/knowhow/vhdl/simple-ram-model/

References

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, ARM Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, arm edition, 2017.