Raisin64 Documentation

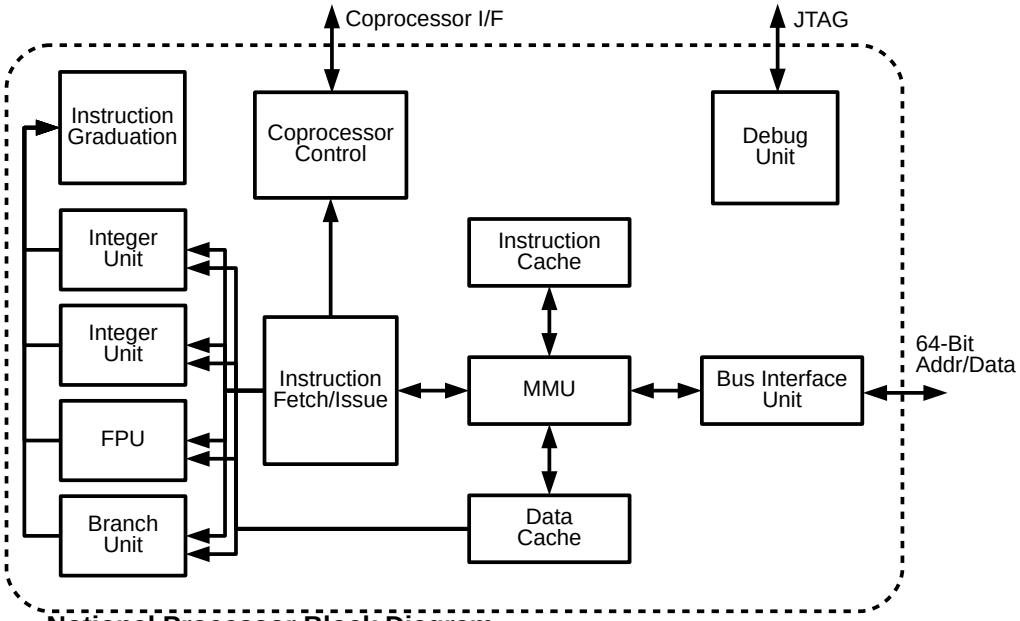
Release 0.1

Christopher Parish

Oct 20, 2018

CONTENTS:

Raisin64 (*RISC Architecture with In-order Superscalar INterlocked-pipeline*) is a pure 64-bit CPU design created as part of an educational project. Architecturally similar to the MIPS R10000 and POWER3, Raisin64 is a superscalar design that employs multiple specialized pipelines for integer operations, floating point, load/store, etc. Unlike most superscalar designs, Raisin64 does not re-order instructions but instead provides a larger architectural register file of 64x64-bit registers.



Notional Processor Block Diagram

Major features of the Raisin64 include:

- **Bits:** 64-bit
- **Design:** RISC
- **Type:** Register-Register
- **Branching:** Condition Code
- **Endianness:** Big
- **Page Size:** 16KB Fixed
- **Virtual Address Size:** 47-Bits
- **Page Table:** Three Level
- **Registers:** 61 (R0 = 0)

CHAPTER

ONE

RAISIN64 CPU

CHAPTER

TWO

CODE SNIPPETS AND SOFTWARE

2.1 Handling Interrupts

2.2 Initializing the MMU

CHAPTER

THREE

TOOLS

3.1 Assembler

3.2 Debugging

3.2.1 Getting OpenOCD

CHAPTER

FOUR

NEXYS 4 DDR REFERENCE IMPLEMENTATION

4.1 SoC Peripherals

4.2 Required Hardware

4.3 Synthesizing the Core

CHAPTER

FIVE

REFERENCE INDEX

5.1 Raisin64 Instruction Set

5.1.1 Overview

The Raisin64's instruction set draws heavily from MIPS with some concepts graciously borrowed from ARM as well. While the programmer's model and instruction set are decoupled from the underlying microarchitecture of the specific implementation, it was nonetheless decided to design the instructions such that a hardwired control unit (see TODO:ref:*Instruction Decode*) could process and set the appropriate signals.

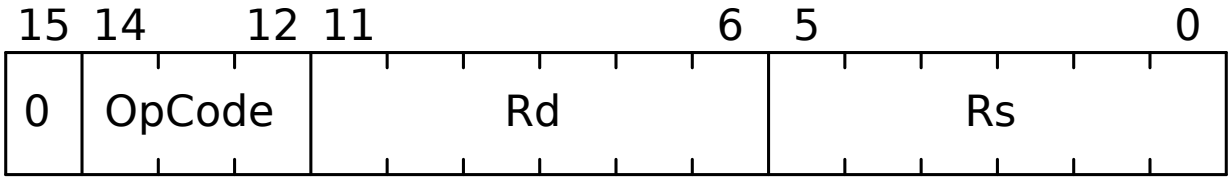
Instructions are variable length (16-64) bit, and some have multiple forms like the *ADD Instruction*. When an instruction has multiple encodings, the opcode is usually the same between the alternate length versions of that instruction, but in all cases the processor expands the 16 and 32-bit versions of the instruction into their canonical 64-bit form, which has a regular encoding. The general instruction formats and opcodes are described below.

But Why?

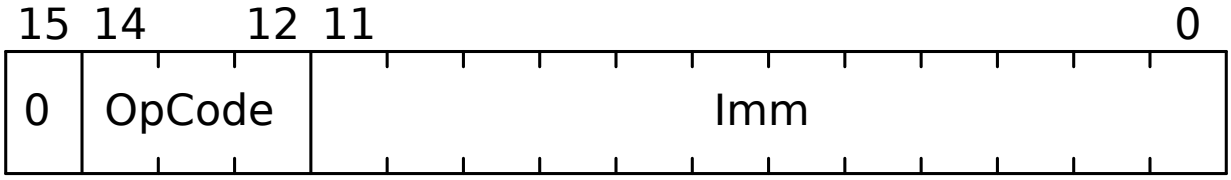
There is a natural appeal to 64 registers on a 64-bit machine. This means 6 bits are needed in the instruction format to address each register. While 64-bit instructions allow this and efficient loading of immediate values, they waste program space more often than not. Variable length instructions are a good compromise to avoid the size penalty when not necessary.

5.1.2 Instruction Format

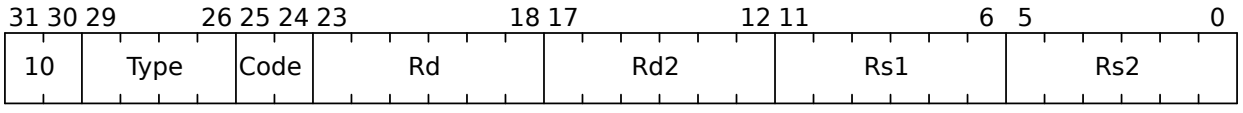
There are 6 instruction formats in Raisin64



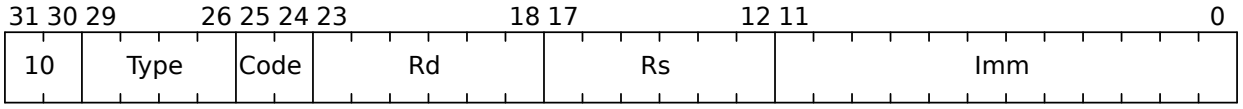
Size



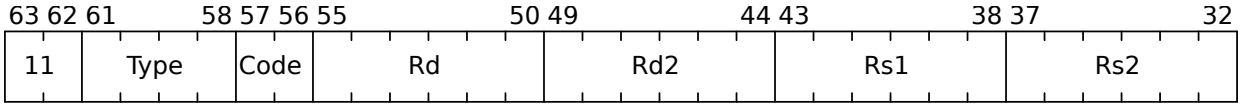
Size



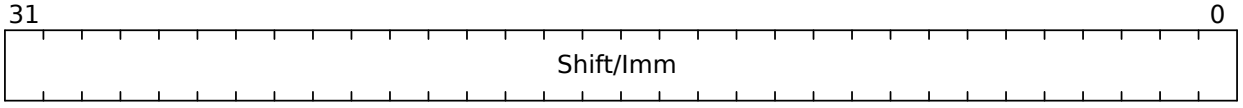
Size

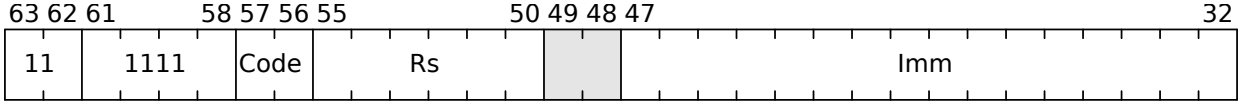


Size

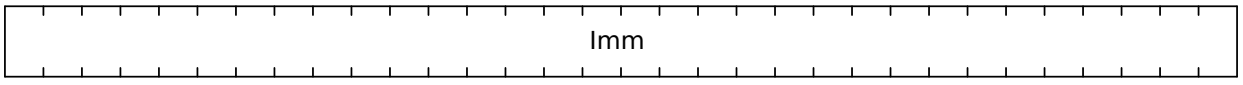


Size





Size Type



5.1.3 Instructions

ADD - Integer Add

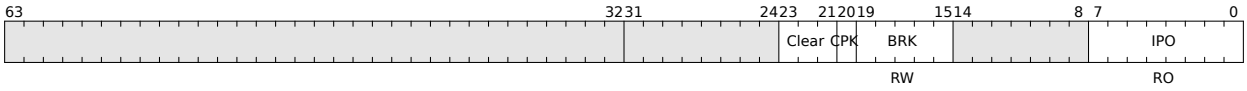


Fig. 1: ADD

ADDI - Integer Add Immediate

AND - Bitwise AND

ANDI - Bitwise AND Immediate

BE - Branch if Even

BO - Branch if Odd

DIV - Integer Divide

DIVI - Integer Divide Immediate

J - Jump

JAL - Jump and Link

JALI - Jump and Link Immediate

JI - Jump Immediate

L16 - Load 16-bit

L16S - Load 16-bit Sign-Extend

L32 - Load 32-bit

L32S - Load 32-bit Sign-Extend

L8 - Load 8-bit

L8S - Load 8-bit Sign-Extend

LW - Load 64-bit Word

MUL - Integer Multiply

MUL - Integer Multiply Immediate

NOR - Bitwise NOR

NORI - Bitwise NOR Immediate

OR - Bitwise OR

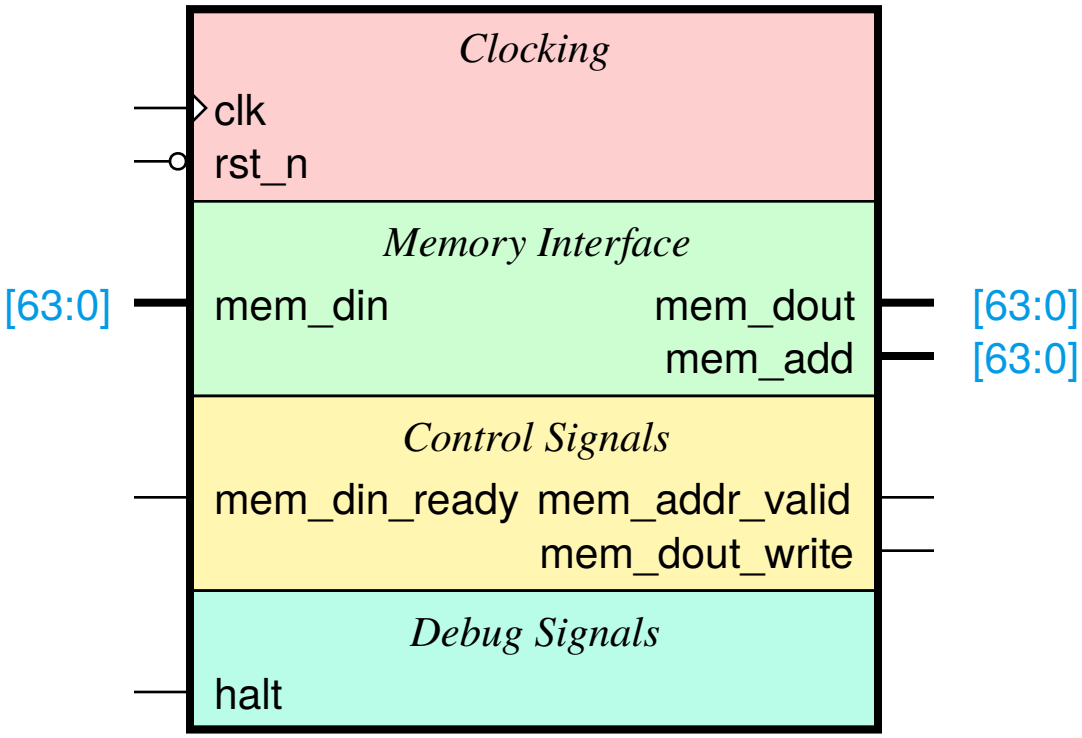


Fig. 2: Raisin64.v

XORI - Bitwise XOR Immediate

5.2 Verilog Module Index

5.2.1 Raisin64.v

```
/*
 * Raisin64 CPU
 */

module raisin64 (
    //# {{clocks|Clocking}}
    input clk,
    input rst_n,

    //# {{data|Memory Interface}}
    input[63:0] mem_din,
    output[63:0] mem_dout,
    output[63:0] mem_add,

    //# {{control|Control Signals}}
    output mem_addr_valid,
    output mem_dout_write
    input mem_din_ready,

    //# {{debug|Debug Signals}}
```

(continues on next page)

(continued from previous page)

```
21      input halt);
22
23  endmodule
```