Raisin64 Documentation

Release 0.1

Christopher Parish

Oct 20, 2018

CONTENTS:

1 Raisin64 CPU **3**
- 1.1 Overview . . . 3
- 1.2 Pipeline Stages . . . 3
- 1.3 Caches . . . 3
- 1.4 MMU . . . 3
- 1.5 Interrupt Unit . . . 3
- 1.6 Debug Unit . . . 3

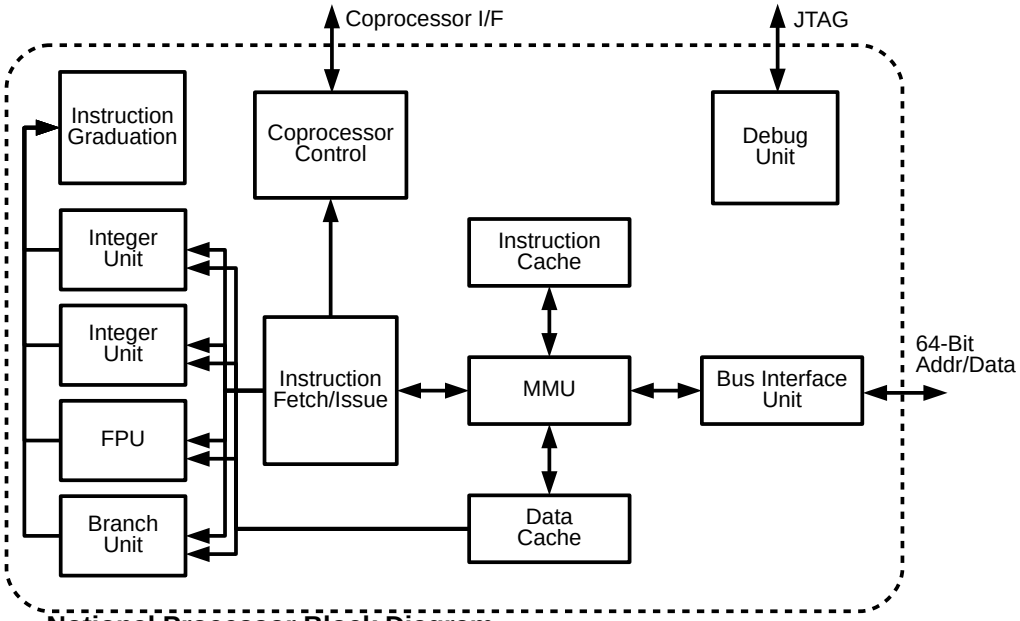
2 Code Snippets and Software **5**
- 2.1 Handling Interrupts . . . 5
- 2.2 Initializing the MMU . . . 5

3 Tools **7**
- 3.1 Assembler . . . 7
- 3.2 Debugging . . . 7

4 Nexys 4 DDR Reference Implementation **9**
- 4.1 SoC Peripherals . . . 9
- 4.2 Required Hardware . . . 9
- 4.3 Synthesizing the Core . . . 9

5 Reference Index **11**
- 5.1 Raisin64 Instruction Set . . . 11
- 5.2 Verilog Module Index . . . 18

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

1.1 Overview

1.2 Pipeline Stages

1.3 Caches

1.4 MMU

1.5 Interrupt Unit

1.6 Debug Unit

CHAPTER

TWO

CODE SNIPPETS AND SOFTWARE

2.1 Handling Interrupts

2.2 Initializing the MMU

CHAPTER

THREE

TOOLS

- *Assembler*
- *Debugging*
  - *Getting OpenOCD*

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

- *Overview*
- *Instruction Format*
- *Unit/Op Table*
  - *16R - 16-bit Register Format*
  - *16I - 16-bit Immediate Format*
  - *32R - 32-bit Register Format*
  - *32I - 32-bit Immediate Format*
  - *64S - 64-bit Standard Format*
  - *64J - 64-bit Jump Format*
- *Instructions*

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, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

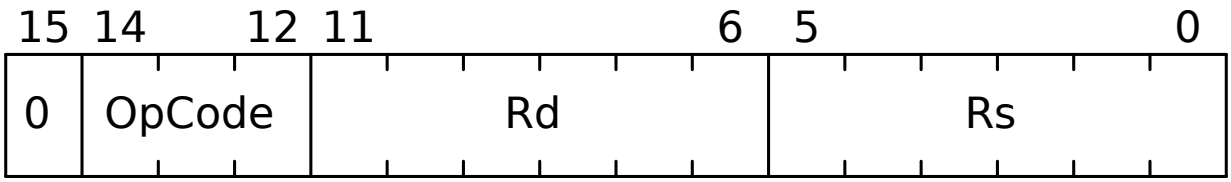
Comparing the 16, 32, and 64-bit formats, the smaller instructions contain those instructions which whill fit in the reduced number of bits. The larger instruction formats are a superset of the smaller ones, and whenever an instruction is available in a smaller format, it is available in all larger formats. For example, ADDI is available in 16, 32, and 64-bit instruction size, with the permitted size of the immediate growing as the instruction grows.

The 32 and 64-bit instruction formats share the same Unit/Op numbers, which are effectively the OpCode. The Unit number represents the type of operation while the Op indicates the specific operation requested.

5.1.3 Unit/Op Table

R/I	Unit	Op
0/1	0 - Integer Math	0 - ADD 1 - SUB 2 - MUL 3 - DIV
	1 - Compare/Set	0 - SEQ 1 - SLE 2 - SLT 3 - SNE
	2 - Shift	0 - SLL 1 - SRA 2 - SRL
	3 - Bitwise Op	0 - AND 1 - NOR 2 - OR 3 - XOR
0	4 - Reserved	
	5 - Reserved	
	6 - Reserved	
1	4 - Regular Load	0 - LW 1 - L32 2 - L16 3 - L8
	5 - Sign-Extend Load	0 - SW 1 - S32 2 - S16 3 - S8
	6 - Store	0 - SW 1 - S32 2 - S16 3 - S8
0	7 - Jump/Special	
		0 - SYSCALL 1 - Reserved 2 - JAL 3 - J

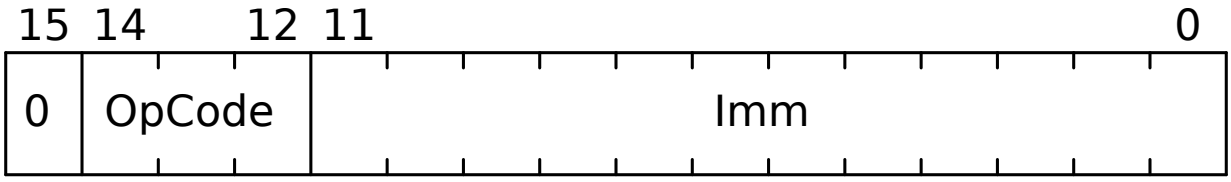
16R - 16-bit Register Format



Size

The 16-bit regsiter format is a compact expression of select instructions operating with one source and one destination register. Instructions normally operating on three registers, such as ADD, instead operate in 2-register mode (i.e. Rd = Rd + Rs).

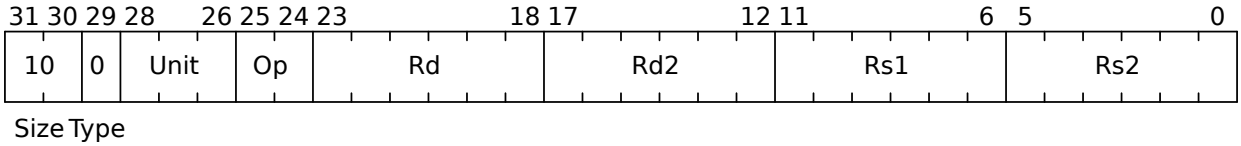
16I - 16-bit Immediate Format



Size

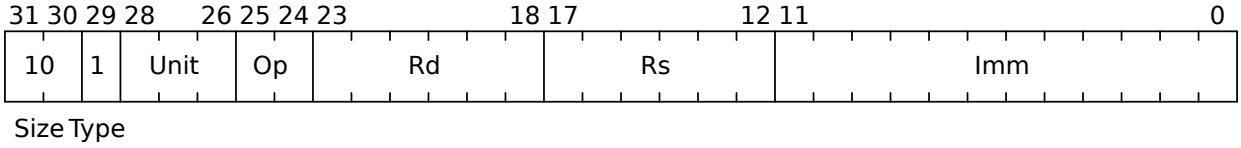
The 16-bit immediate format is used only for ADDI and SUBI, allowing for small increment and decrement operations in a compact format.

32R - 32-bit Register Format



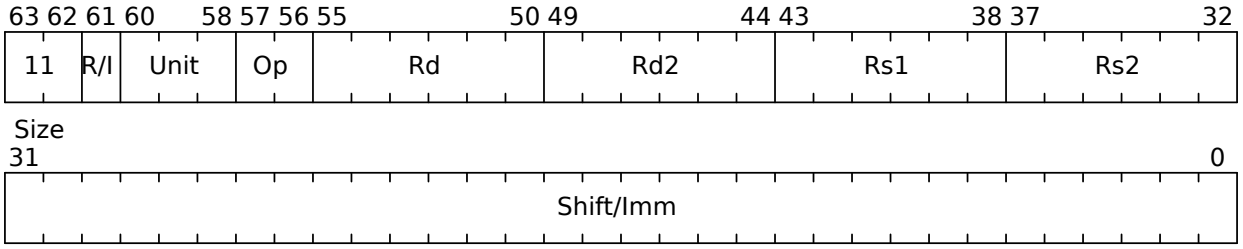
All register type insturctions in the Raisin64 are available in 32R format.

32I - 32-bit Immediate Format



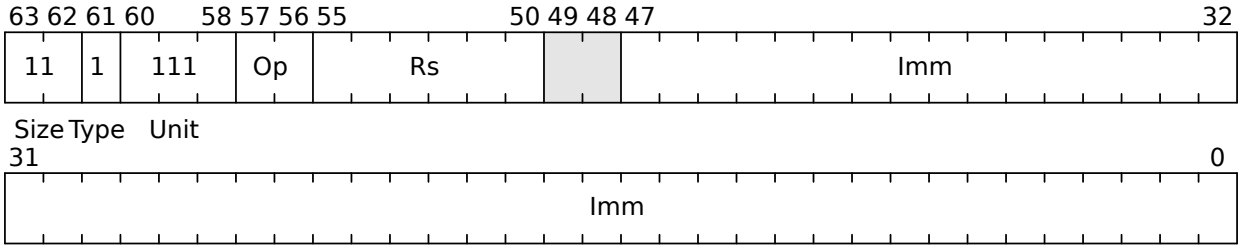
With the exception of JI, JALI, DIVI, and MULI, all immediate type instructions in the Raisin64 are available in the 32I format albeit with a reduced 12-bit immediate value.

64S - 64-bit Standard Format



All register and immediate type instructions (with the expcetion of branch insturctions) from the 32-bit instruction formats are available in the unified 64S format. This 64-bit format has space for 4 registers (allowing for instructions like MUL) in addition to 32-bits of immediate data (for shifting and bitwise operations).

64J - 64-bit Jump Format



A special jump format for large displacement BE, BO, JI, and JALI, the 64J format allows for a full 48-bit displacement/jump, sufficient to cover the entire virtual address space of the Raisin64.

5.1.4 Instructions

ADD - Integer Add

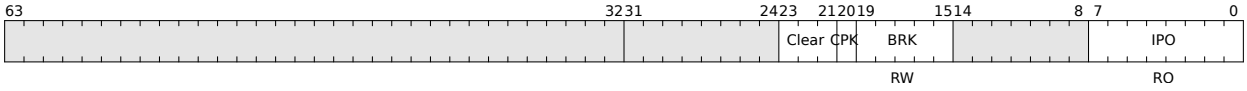


Fig. 1: ADD

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

ORI - Bitwise OR Immediate

S16 - Store 16-bit

S32 - Store 32-bit

S8 - Store 8-bit

SEQ - Set 1 if Equal

SEQI - Set 1 if Equal Immediate

SLE - Set 1 if Less than or Equal

SLEI - Set 1 if Less than or Equal Immediate

SLL - Shift Left Logical

SLLI - Shift Left Logical Immediate

SLT - Set 1 if Less Than

SLTI - Set 1 if Less Than Immediate

SNE - Set 1 if Not Equal

SNEI - Set 1 if Not Equal Immediate

SRA - Shift Right Arithmetic

SRAI - Shift Right Arithmetic Immediate

SRL - Shift Right Logical

SRLI - Shift Right Logical Immediate

SUB - Integer Subtract

SUBI - Integer Subtract Immediate

SW - Store 64-bit Word

SYSCALL - System Call

XOR - Bitwise XOR

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,
```

(continues on next page)

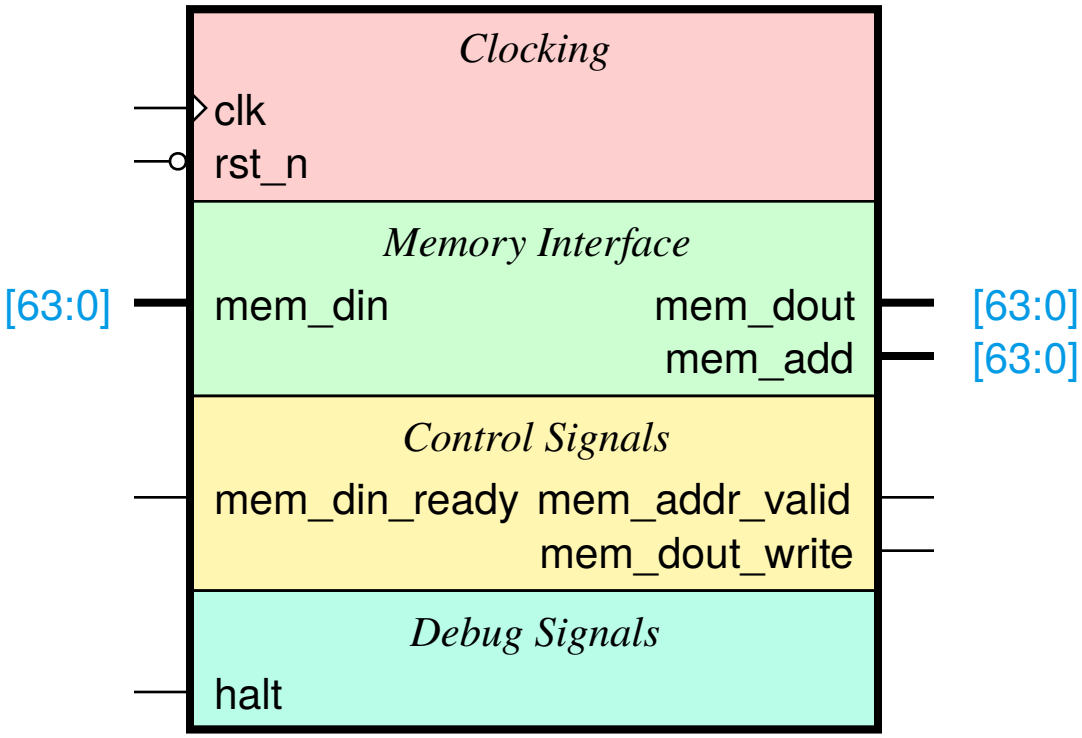


Fig. 2: Raisin64.v

(continued from previous page)

```
17      output mem_dout_write
18      input mem_din_ready,
19
20      //# {{debug|Debug Signals}}
21      input halt);
22
23  endmodule
```