Raisin64 Documentation

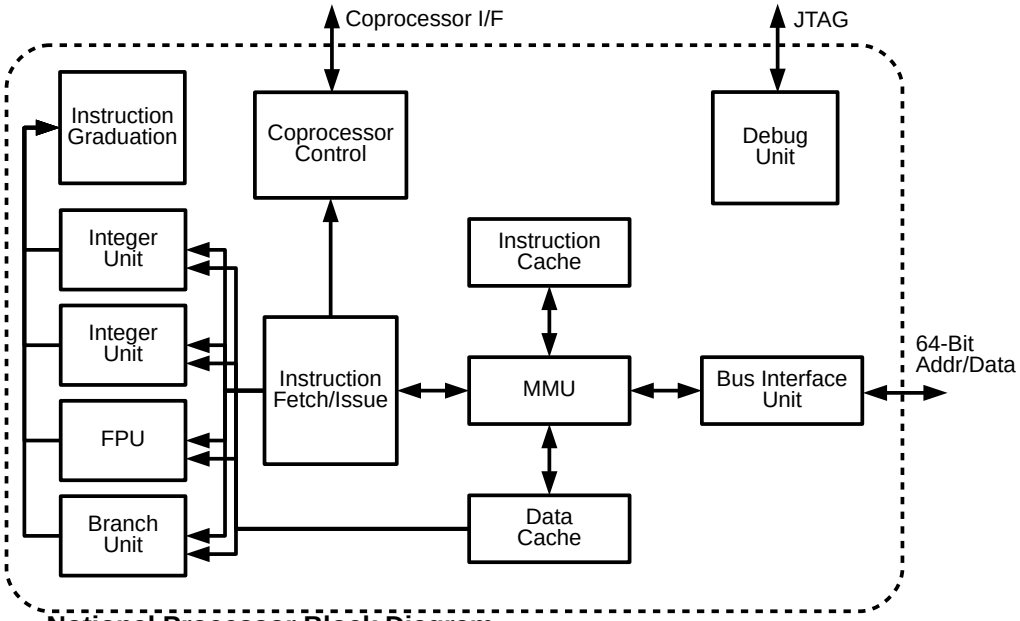
Release 0.1

Christopher Parish

Oct 21, 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

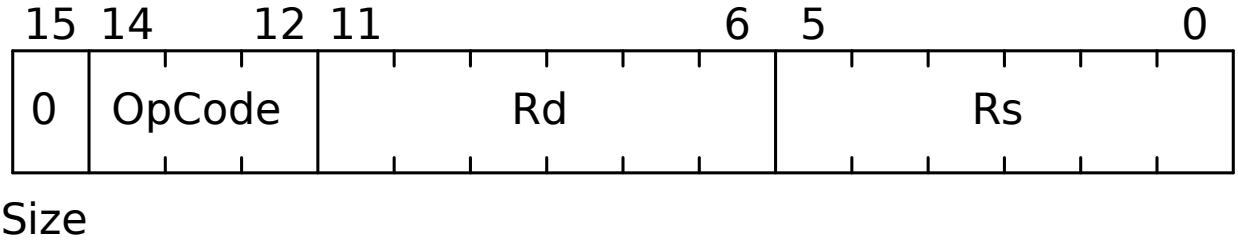
There are 6 instruction formats in Raisin64, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

Comparing the 16, 32, and 64-bit formats, the smaller instructions contain those instructions which will fit in the reduced number of bits. The larger instruction formats are a super-set 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. This conveniently fits into the first 8 bits of the instruction, making the opcode easier to view and manipulate.

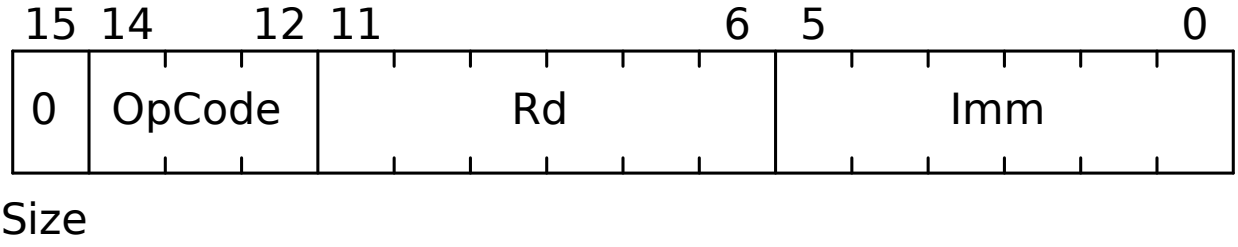
5.1.3 16-bit formats

16R - 16-bit Register Format



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



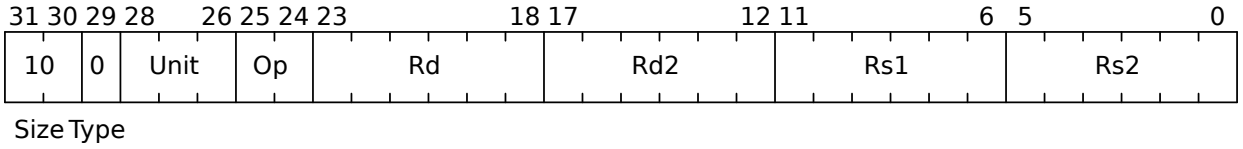
The 16-bit immediate format is used only for ADDI and SUBI, allowing for small increment and decrement operations in a compact format.

16-bit OpCode Table

OpCode	Type
0 - ADD	16R
1 - SUB	16R
2 - ADDI	16I
3 - SUBI	16I
4 - SYSCALL	16R
5 - J	16R
6 - JAL	16R
7 - Reserved	

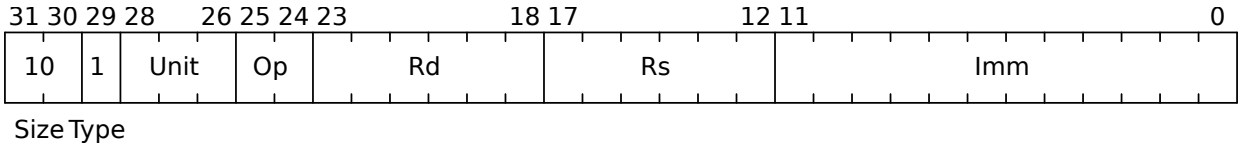
5.1.4 32/64-bit Formats

32R - 32-bit Register Format



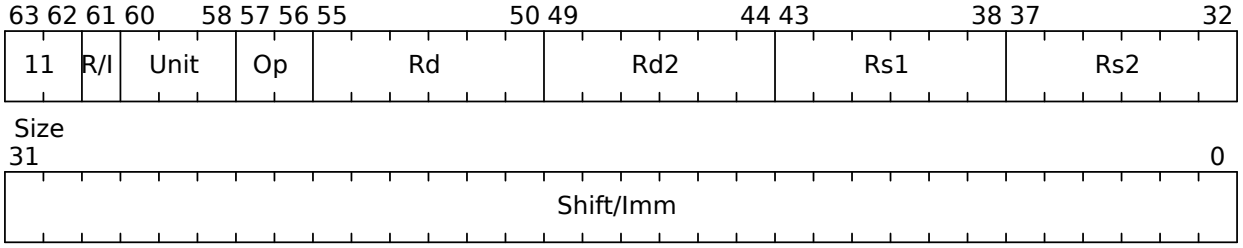
All register type instructions in the Raisin64 are available in 32R format. The only exception of this is LUI, which is formatted as a regsiter type instruction but depends on the 32-bit immediate field of the 64S format, making it not available in 32-bit format.

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.

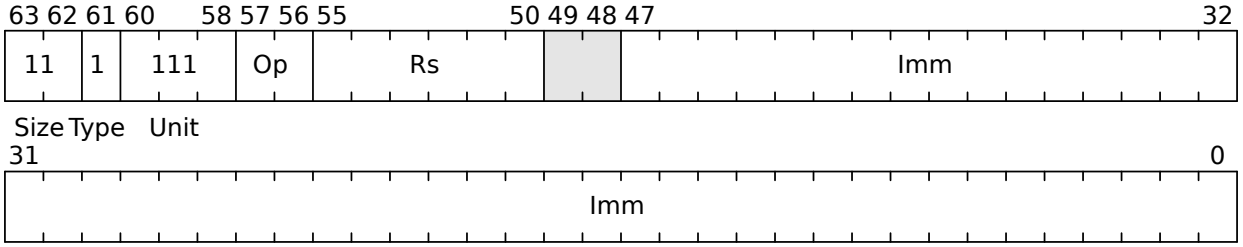
64S - 64-bit Standard Format



All register and immediate type instructions (with the exception of branch instructions) from the 32-bit instruction formats are available in the unified 64S format. When these smaller instructions are encountered by the Raisin64,

they are internally expanded into canonical 64S format before being passed onto the rest of the processor. 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.

32 and 64-bit Unit/Op Table

R/I	Unit	Op
0/1	0 - Integer Math	0 - ADD/ANDI
		1 - SUB/SUBI
		2 - MUL/MULI
		3 - DIV/DIVI
	1 - Compare/Set	0 - SEQ/SEQI
		1 - SLE/SLEI
		2 - SLT/SLTI
		3 - SNE/SNEI
	2 - Shift	0 - SLL/SLLI
		1 - SRA/SRAI
		2 - SRL/SRLI
	3 - Bitwise Op	0 - AND/ANDI
		1 - NOR/NORI
		2 - OR/ORI
		3 - XOR/XORI
0	4 - Floating Point	
	5 - Unused Extension	
	6 - Unused Extension	
1	4 - Regular Load	0 - LW
		1 - L32
		2 - L16
		3 - L8
	5 - Sign-Extend Load	0 - Reserved
		1 - L32S
		2 - L16S
		3 - L8S
	6 - Store	0 - SW
		1 - S32
		2 - S16

Continued on next page

Table 1 – continued from previous page

R/I	Unit	Op
		3 - S8
0	7 - Jump/Special	0 - SYSCALL
		1 - LUI
		2 - JAL
		3 - J
1	7 - Jump Immediate	0 - BE
		1 - BO
		2 - JALI
		3 - JI

5.1.5 Instructions

ADD - Integer Add

Adds registers `Rs1` and `Rs2`, placing the result in `Rd`.

Usage

```
add Rd, Rs1, Rs2
```

Operation

```
Rd = Rs1 + Rs2;
advance_pc();
```

Encoding

Type 0

Unit 0

Op 0

16-bit Opcode 0x0

32-bit Opcode 0x80

64-bit Opcode 0xC0

ADDI - Integer Add Immediate

Adds registers `Rs1` and a sign-extended immediate value, placing the result in `Rd`.

Usage

```
addi Rd, Rs1, imm
```

Operation

```
Rd = Rs1 + sign_extend(imm);
advance_pc();
```

Encoding

Type 1

Unit 0

Op 0

16-bit Opcode 0x2

32-bit Opcode 0xA0

64-bit Opcode 0xE0

AND - Bitwise AND

Bitwise ANDs registers `Rs1` and `Rs2`, placing the result in `Rd`.

Usage

```
and Rd, Rs1, Rs2
```

Operation

```
Rd = Rs1 & Rs2;
advance_pc();
```

Encoding

Type 0

Unit 3

Op 0

16-bit Opcode NONE

32-bit Opcode 0x8C

64-bit Opcode 0xCC

ANDI - Bitwise AND Immediate

Bitwise ANDs register `Rs1` and an immediate value, placing the result in `Rd`.

Usage

```
andi Rd, Rs1, imm
```

Operation

```
Rd = Rs1 & imm;
advance_pc();
```

Encoding

Type 1

Unit 3

Op 0

16-bit Opcode NONE

32-bit Opcode 0xAC

64-bit Opcode 0xEC

BE - Branch if Even

Branches if the `Rs` register is even. An unconditional branch can be accomplished by using `$r0` as zero is even. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump.

Usage

```
be Rs, imm
```

Operation

```
if(is_even(Rs))
   pc = pc+(imm<<1);
else
   advance_pc();
```

Encoding

Type 1

Unit 7

Op 0

16-bit Opcode NONE

32-bit Opcode 0xBC

64-bit Opcode 0xFC

BO - Branch if Odd

Branches if the `Rs` register is odd. This is useful in conjunction with the set type instructions (SEQ, SLE, etc.) to branch on a positive result as 1 is odd. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump.

Usage

```
bo Rs, imm
```

Operation

```
if(is_odd(Rs))
   pc = pc+(imm<<1);
else
   advance_pc();
```

Encoding

Type 1

Unit 7

Op 1

16-bit Opcode NONE

32-bit Opcode 0xBD

64-bit Opcode 0xFD

DIV - Integer Divide

Divides registers `Rs1` by `Rs2` and places the quotient in `Rd` and the remainder in `Rd2`.

Usage

```
div Rd, Rd2, Rs1, Rs2
```

Operation

```
Rd = Rs1 / Rs2;
Rd2 = Rs1 % Rs2;
advance_pc();
```

Encoding

Type 0

Unit 0

Op 3

16-bit Opcode NONE

32-bit Opcode 0x83

64-bit Opcode 0xC3

DIVI - Integer Divide Immediate

Divides register `Rs1` by an immediate value and places the quotient in `Rd` and the remainder in `Rd2`. Because this instruction requires three registers and an immediate value, it is only available in 64S format.

Usage

```
div Rd, Rd2, Rs1, imm
```

Operation

```
Rd = Rs1 / imm;
Rd2 = Rs1 % imm;
advance_pc();
```

Encoding

Type 1

Unit 0

Op 3

16-bit Opcode NONE

32-bit Opcode NONE

64-bit Opcode 0xE3

J - Jump

Unconditional jump to the instruction in `Rs`. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump.

Usage

```
j Rs
```

Operation

```
pc = Rs;
```

Encoding

Type 0

Unit 7

Op 3

16-bit Opcode 0x5

32-bit Opcode 0x9F

64-bit Opcode 0xDF

JAL - Jump and Link

Unconditional jump to the instruction in `Rs`, placing the return address in $63. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump.

Usage

```
jal Rs
```

Operation

```
r63 = next_pc();
pc = Rs;
```

Encoding

Type 0

Unit 7

Op 2

16-bit Opcode 0x6

32-bit Opcode 0x9E

64-bit Opcode 0xDE

JALI - Jump and Link Immediate

Unconditional jump to the immediate value, placing the return address in $63. The top 16 bits of the jump destination address are taken from the current program counter. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump. Due to the size of the immediate value, JALI is only available in 64J format.

Usage

```
jali imm
```

Operation

```
r63 = pc + 8;
pc = (pc & 0xffff000000000000) | imm<<1;
```

Encoding

Type 1

Unit 7

Op 2

16-bit Opcode NONE

32-bit Opcode NONE

64-bit Opcode 0xFE

JI - Jump Immediate

Unconditional jump to the immediate value. The top 16 bits of the jump destination address are taken from the current program counter. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump. Due to the size of the immediate value, JI is only available in 64J format.

Usage

```
ji imm
```

Operation

```
pc = (pc & 0xffff000000000000) | imm<<1;
```

Encoding

Type 1

Unit 7

Op 3

16-bit Opcode NONE

32-bit Opcode NONE

64-bit Opcode 0xFF

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

LUI - Load Upper Immediate

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

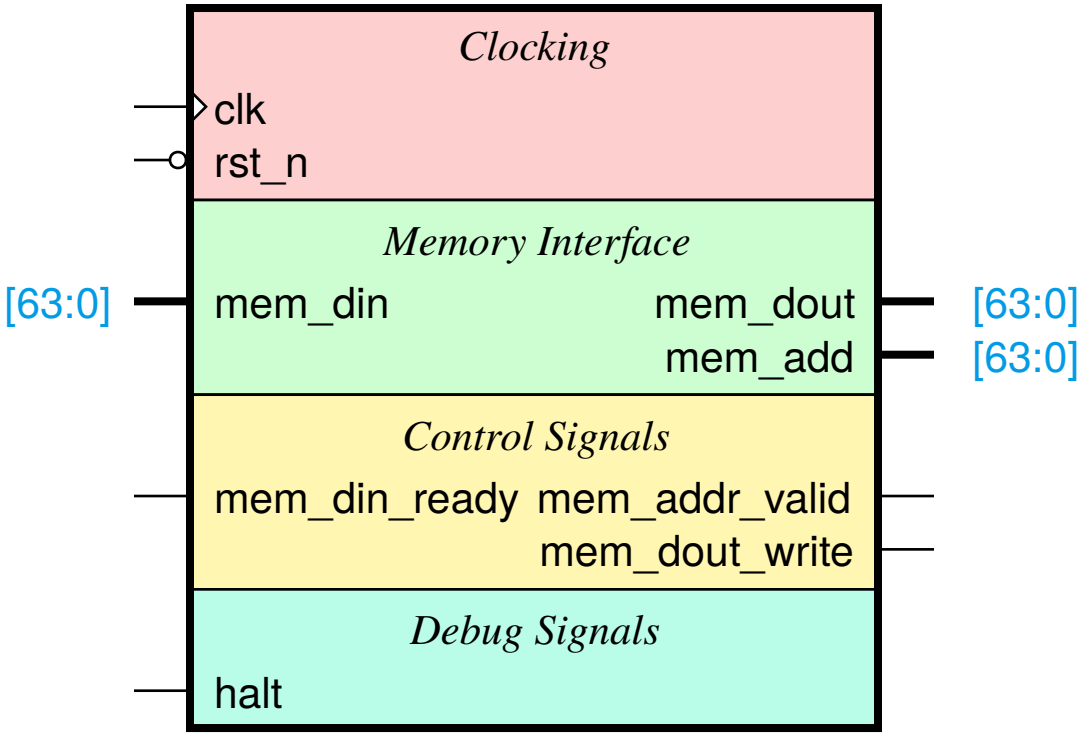


Fig. 1: Raisin64.v

```
/*
 * 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}}
    input halt);

endmodule
```