

---

# **Raisin64 Documentation**

***Release 0.1***

**Christopher Parish**

**Nov 03, 2018**

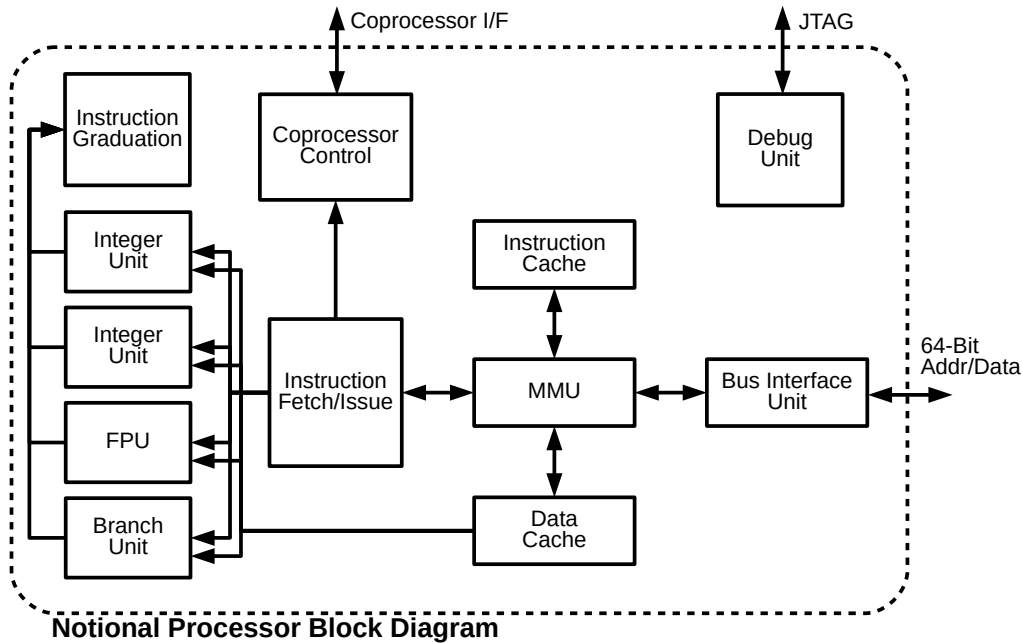


**CONTENTS:**

<b>1</b>	<b>Raisin64 CPU</b>	<b>3</b>
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
<b>2</b>	<b>Code Snippets and Software</b>	<b>5</b>
2.1	Handling Interrupts . . . . .	5
2.2	Initializing the MMU . . . . .	5
<b>3</b>	<b>Tools</b>	<b>7</b>
3.1	Assembler . . . . .	7
3.2	Debugging . . . . .	7
<b>4</b>	<b>Nexys 4 DDR Reference Implementation</b>	<b>9</b>
4.1	SoC Peripherals . . . . .	9
4.2	Required Hardware . . . . .	9
4.3	Synthesizing the Core . . . . .	9
<b>5</b>	<b>Reference Index</b>	<b>11</b>
5.1	Raisin64 Instruction Set . . . . .	11
5.2	Verilog Module Index . . . . .	41



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.



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)



**RAISIN64 CPU**

**1.1 Overview**

**1.2 Pipeline Stages**

**1.3 Caches**

**1.4 MMU**

**1.5 Interrupt Unit**

**1.6 Debug Unit**





## CODE SNIPPETS AND SOFTWARE

### 2.1 Handling Interrupts

### 2.2 Initializing the MMU



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

## **3.1 Assembler**

## **3.2 Debugging**

### **3.2.1 Getting OpenOCD**



## **NEXYS 4 DDR REFERENCE IMPLEMENTATION**

### **4.1 SoC Peripherals**

### **4.2 Required Hardware**

### **4.3 Synthesizing the Core**



## REFERENCE INDEX

### 5.1 Raisin64 Instruction Set

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

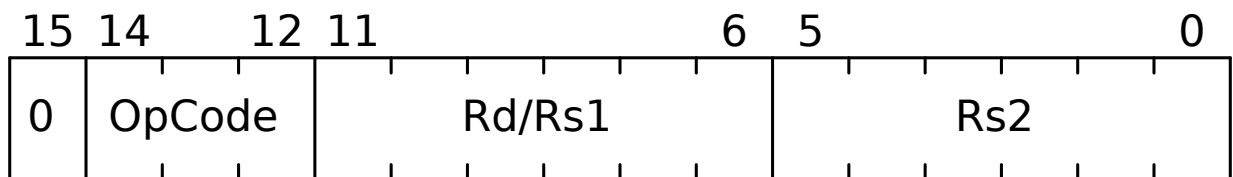
There are 6 instruction formats in Raisin64, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

Comparing the 16, 32, and 64-bit formats, the smaller instructions contain those instructions which 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

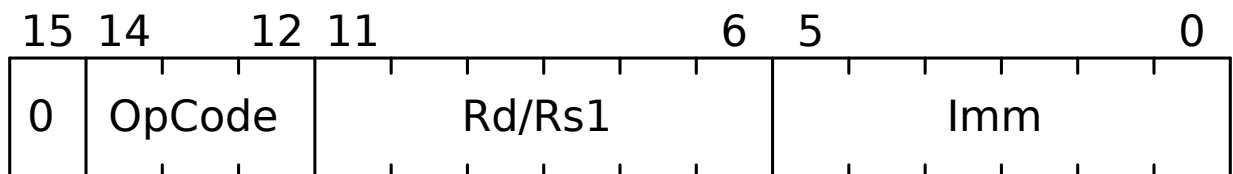
#### 16R - 16-bit Register Format



#### Size

The 16-bit register 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.

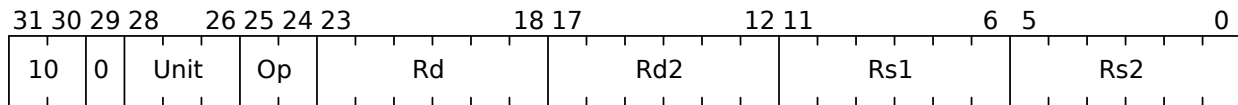


## 16-bit OpCode Table

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

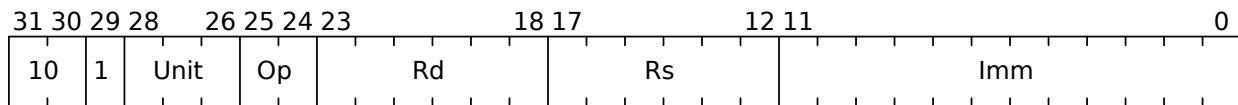
### 32R - 32-bit Register Format



Size Type

All register type instructions in the Raisin64 are available in 32R format. The only exception of this is the F\* FPU call, which also uses the immediate field of the 64S format.

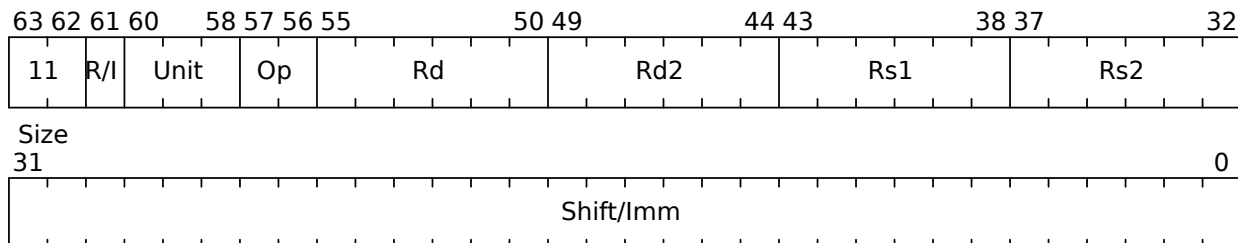
### 32I - 32-bit Immediate Format



Size Type

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

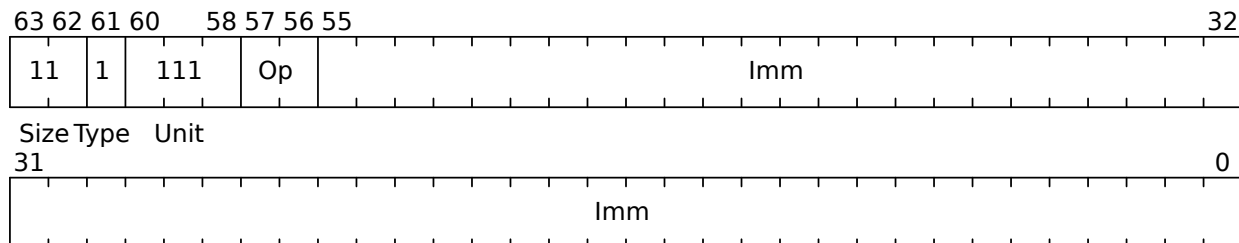
### 64S - 64-bit Standard Format



All register and immediate type instructions (except the immediate type branch and jump instructions) are available in the unified 64S format. When smaller width 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, bitwise, and ordinary immediate operations).

### 64J - 64-bit Jump Format



A special jump format for large displacement JI and JALI, the 64J format allows for a full 56-bit unsigned jump, more than sufficient to cover the entire virtual address space of the Raisin64.

### 32 and 64-bit Unit/Op Table

R/I	Unit	Op
0	0 - Basic Integer Math	0 - ADD
		1 - SUB
	1 - Compare/Set	0 - SLT
		1 - SLTU
		2 - SGT
		3 - SGTU
	2 - Shift	0 - SLL
		1 - SRA
		2 - SRL
	3 - Bitwise Op	0 - AND
		1 - NOR
		2 - OR
		3 - XOR
	4 - Advanced Integer Math	0 - MUL
		1 - MULU
		2 - DIV
		3 - DIVU
	5 - Reserved	
	6 - Reserved	
	7 - Jump/Special	0 - SYSCALL
		1 - F* (FPU Call) <sup>12</sup>
		2 - JAL
		3 - J
1	0 - Basic Integer Math	0 - ADDI
		1 - SUBI
	1 - Compare/Set	0 - SLTI
		1 - SLTIU
		2 - SGTI
		3 - SGTIU
	2 - Shift	0 - SLLI

Continued on next page

Table 1 – continued from previous page

R/I	Unit	Op
		1 - SRAI
		2 - SRLI
	3 - Bitwise Op	0 - ANDI
		1 - NORI
		2 - ORI
		3 - XORI
	4 - Regular Load	0 - LW
		1 - L32
		2 - L16
		3 - L8
	5 - Sign-Extend Load	0 - LUI <sup>1</sup>
		1 - L32S
		2 - L16S
		3 - L8S
	6 - Store	0 - SW
		1 - S32
		2 - S16
		3 - S8
	7 - Jump Immediate	0 - BEQ
		1 - BEQAL
		2 - JALI <sup>1</sup>
		3 - JI <sup>1</sup>

## 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

<sup>1</sup> 64-bit format only

<sup>2</sup> The F\* instruction uses the immediate field of 64S to request a specific enumerated service from the FPU. These instructions are documented TODO::ref::here.

**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

## BEQ - Branch if Equal

If the `Rs` register is equal to the `Rd` register, the program branches by the signed immediate displacement. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump.

---

**Tip:** An unconditional branch can be accomplished by comparing `r0` with itself.

---

## Usage

```
beq Rd, Rs, imm
```

## Operation

```
if(Rd == 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

## BEQAL - Branch if Equal And Link

If the `Rs` register is equal to the `Rd` register, the program branches by the signed immediate displacement. The address of the next linear instruction is placed as a return address in `r63`. As instructions must be aligned to 16-bit boundaries, the immediate value is left shifted by 1 before the jump.

---

**Tip:** An unconditional branch can be accomplished by comparing `r0` with itself.

---

## Usage

```
beqal Rd, Rs, imm
```

## Operation

```
if(Rd == Rs)
    r63 = next_pc();
    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`, treating operands as 2's complement signed.

## Usage

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

## Operation

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

## Encoding

**Type** 0

**Unit** 4

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0x92

**64-bit Opcode** 0xD2

## DIVU - Unsigned Integer Divide

Divides registers `Rs1` by `Rs2`, and places the quotient in `Rd` and the remainder in `Rd2`, treating operands as unsigned.

## Usage

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

## Operation

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

## Encoding

**Type** 0  
**Unit** 4  
**Op** 3  
**16-bit Opcode** NONE  
**32-bit Opcode** 0x93  
**64-bit Opcode** 0xD3

## F\* - FPU Call

### Usage

```
todo
```

## Operation

```
todo;
advance_pc();
```

## Encoding

**Type** 0  
**Unit** 7  
**Op** 1  
**16-bit Opcode** NONE  
**32-bit Opcode** NONE  
**64-bit Opcode** 0xDD

## 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 `r63`. 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 `r63`. The top 8 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 & 0xff00000000000000) | 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 8 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 & 0xff00000000000000) | 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

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 4

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0xB2

**64-bit Opcode** 0xF2

## L16S - Load 16-bit Sign-Extend

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 5

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0xB6

**64-bit Opcode** 0xF6

## L32 - Load 32-bit

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 4

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0xB1

**64-bit Opcode** 0xF1

## L32S - Load 32-bit Sign-Extend

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 5

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0xB5

**64-bit Opcode** 0xF5

## L8 - Load 8-bit

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 4

**Op** 3

**16-bit Opcode** NONE

**32-bit Opcode** 0xB3

**64-bit Opcode** 0xF3

## L8S - Load 8-bit Sign-Extend

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 5

**Op** 3

**16-bit Opcode** NONE

**32-bit Opcode** 0xB7

**64-bit Opcode** 0xF7

## LUI - Load Upper Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 5

**Op** 0

**16-bit Opcode** NONE

**32-bit Opcode** NONE

**64-bit Opcode** 0xF4

## LW - Load 64-bit Word

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 4

**Op** 0

**16-bit Opcode** NONE

**32-bit Opcode** 0xB0

**64-bit Opcode** 0xF0

## MUL - Integer Multiply

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 4

**Op** 0

**16-bit Opcode** NONE

**32-bit Opcode** 0x90

**64-bit Opcode** 0xD0

## MULU - Unsigned Integer Multiply

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 4

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0x91

**64-bit Opcode** 0xD1

## NOR - Bitwise NOR

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 3

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0x8D

**64-bit Opcode** 0xCD

## NORI - Bitwise NOR Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```



## Encoding

**Type** 1

**Unit** 3

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0xAD

**64-bit Opcode** 0xED

## OR - Bitwise OR

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 3

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0x8E

**64-bit Opcode** 0xCE

## ORI - Bitwise OR Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 3

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0xAE

**64-bit Opcode** 0xEE

## S16 - Store 16-bit

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 6

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0xBA

**64-bit Opcode** 0xFA

## S32 - Store 32-bit

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 6

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0xB9

**64-bit Opcode** 0xF9

## S8 - Store 8-bit

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 6

**Op** 3

**16-bit Opcode** NONE

**32-bit Opcode** 0xBB

**64-bit Opcode** 0xFB

## SGT - Set 1 if Greater Than

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 1

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0x86

**64-bit Opcode** 0xC6

## SGTI - Set 1 if Greater Than Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 1

**Op** 2

**16-bit Opcode** NONE

**32-bit Opcode** 0xA6

**64-bit Opcode** 0xE6

## SGTIU - Set 1 if Greater Than Immediate Unsigned

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 1

**Op** 3

**16-bit Opcode** NONE

**32-bit Opcode** 0xA7

**64-bit Opcode** 0xE7

## SGTU - Set 1 if Greater Than Unsigned

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 1

**Op** 3

**16-bit Opcode** NONE

**32-bit Opcode** 0x87

**64-bit Opcode** 0xC7

## SLL - Shift Left Logical

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0  
**Unit** 2  
**Op** 0  
**16-bit Opcode** NONE  
**32-bit Opcode** 0x88  
**64-bit Opcode** 0xC8

## SLLI - Shift Left Logical Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1  
**Unit** 2  
**Op** 0  
**16-bit Opcode** NONE  
**32-bit Opcode** 0xA8  
**64-bit Opcode** 0xE8

## SLT - Set 1 if Less Than

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 1

**Op** 0

**16-bit Opcode** NONE

**32-bit Opcode** 0x84

**64-bit Opcode** 0xC4

## SLTI - Set 1 if Less Than Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 1

**Op** 0

**16-bit Opcode** NONE

**32-bit Opcode** 0xA4

**64-bit Opcode** 0xE4

## SLTIU - Set 1 if Less Than Immediate Unsigned

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1  
**Unit** 1  
**Op** 1  
**16-bit Opcode** NONE  
**32-bit Opcode** 0xA5  
**64-bit Opcode** 0xE5

## SLTU - Set 1 if Less Than Unsigned

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0  
**Unit** 1  
**Op** 1  
**16-bit Opcode** NONE  
**32-bit Opcode** 0x85  
**64-bit Opcode** 0xC5

## SRA - Shift Right Arithmetic

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```



## Encoding

**Type** 0

**Unit** 2

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0x89

**64-bit Opcode** 0xC9

## SRAI - Shift Right Arithmetic Immediate

### Usage

```
todo
```

### Operation

```
todo;
advance_pc();
```

## Encoding

**Type** 1

**Unit** 2

**Op** 1

**16-bit Opcode** NONE

**32-bit Opcode** 0xA9

**64-bit Opcode** 0xE9

## SRL - Shift Right Logical

### Usage

```
todo
```

### Operation

```
todo;
advance_pc();
```

## Encoding

**Type** 0  
**Unit** 2  
**Op** 2  
**16-bit Opcode** NONE  
**32-bit Opcode** 0x8A  
**64-bit Opcode** 0xC8

## SRLI - Shift Right Logical Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1  
**Unit** 2  
**Op** 2  
**16-bit Opcode** NONE  
**32-bit Opcode** 0xAA  
**64-bit Opcode** 0xEA

## SUB - Integer Subtract

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 0

**Op** 1

**16-bit Opcode** 0x1

**32-bit Opcode** 0x81

**64-bit Opcode** 0xC1

## SUBI - Integer Subtract Immediate

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 0

**Op** 1

**16-bit Opcode** 0x3

**32-bit Opcode** 0xA1

**64-bit Opcode** 0xE1

## SW - Store 64-bit Word

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 1

**Unit** 6

**Op** 0

**16-bit Opcode** NONE

**32-bit Opcode** 0xB8

**64-bit Opcode** 0xF8

## SYSCALL - System Call

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0

**Unit** 7

**Op** 0

**16-bit Opcode** 0x4

**32-bit Opcode** 0x9C

**64-bit Opcode** 0xDC

## XOR - Bitwise XOR

### Usage

```
todo
```

### Operation

```
todo;  
advance_pc();
```

## Encoding

**Type** 0  
**Unit** 3  
**Op** 3  
**16-bit Opcode** NONE  
**32-bit Opcode** 0x8F  
**64-bit Opcode** 0xCF

## XORI - Bitwise XOR Immediate

### Usage

```
todo
```

### Operation

```
todo;
advance_pc();
```

## Encoding

**Type** 1  
**Unit** 3  
**Op** 3  
**16-bit Opcode** NONE  
**32-bit Opcode** 0xAF  
**64-bit Opcode** 0xEF

## 5.2 Verilog Module Index

### 5.2.1 Raisin64.v

```

1  /*
2   * Raisin64 CPU
3   */
4
5  module raisin64 (
6      //# {{clocks|Clocking}}
7      input  clk,
8      input  rst_n,
9  
```

(continues on next page)

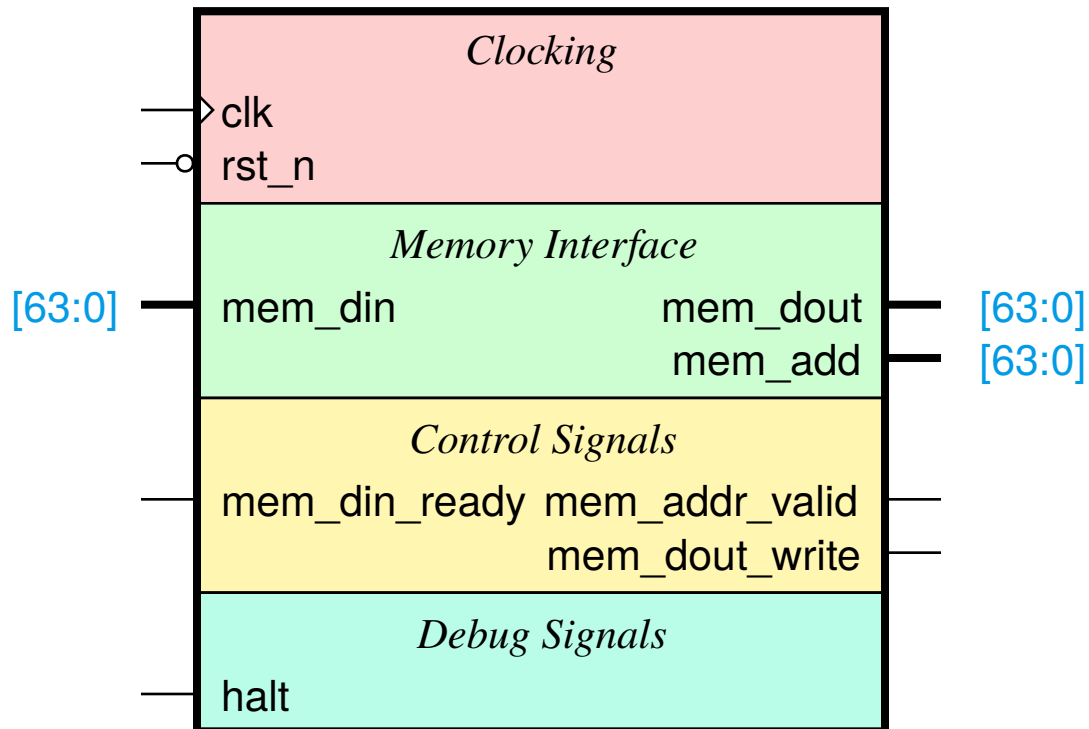


Fig. 1: Raisin64.v

(continued from previous page)

```

10  //# {{data|Memory Interface}}
11  input [63:0] mem_din,
12  output [63:0] mem_dout,
13  output [63:0] mem_add,
14
15  //# {{control|Control Signals}}
16  output mem_addr_valid,
17  output mem_dout_write
18  input mem_din_ready,
19
20  //# {{debug|Debug Signals}}
21  input halt);
22
23  endmodule

```