Raisin64 Documentation

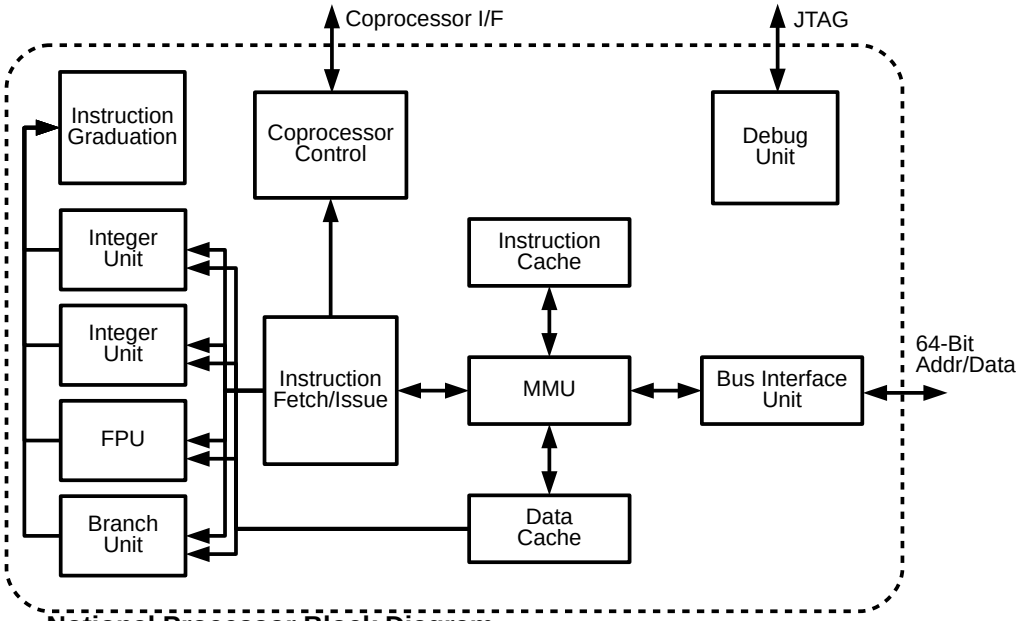
Release 0.1

Christopher Parish

Nov 04, 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

1.1 Overview

1.2 Pipeline Stages

Note: In general, all functional units are coded as combinational logic, with explicit register stages dividing up the processing pipeline. This was done intentionally to allow expermintation and easy combining/splitting of functionality into however many registered stages are necessary to meet timing. The pipeline register modules/files are suffexed with <name>_reg.

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

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

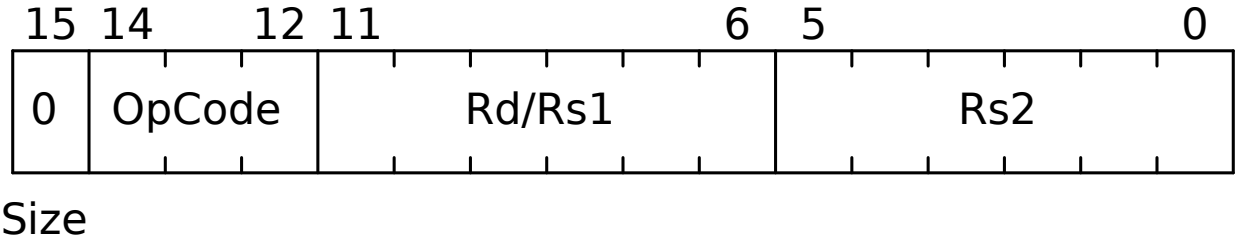
There are 6 instruction formats in Raisin64, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

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.

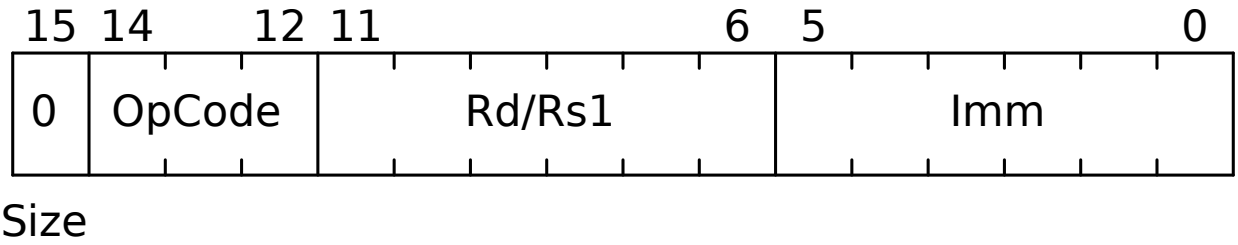
5.1.3 16-bit formats

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



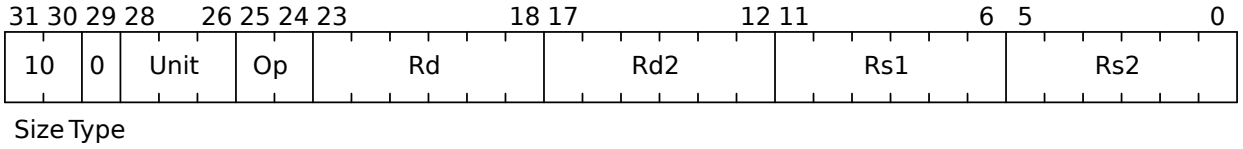
The 16-bit immediate format is used only for ADDI and SUBI, allowing for small increment and decrement operations in a compact format.

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	

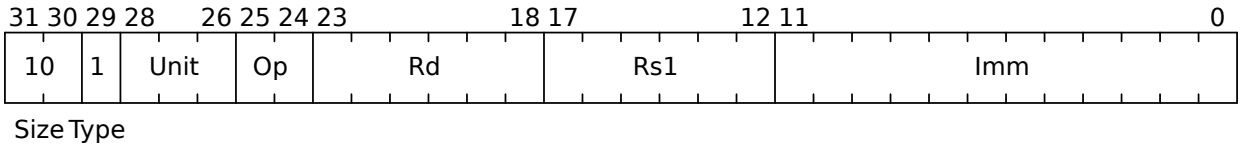
5.1.4 32/64-bit Formats

32R - 32-bit Register Format



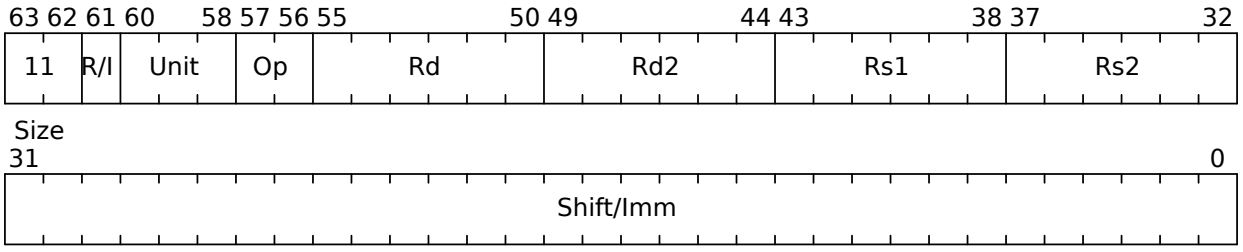
All register type instructions in the Raisin64 are available in 32R format. The only exception of this is the F* FPU call, which also uses the immediate field of the 64S format.

32I - 32-bit Immediate Format



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.

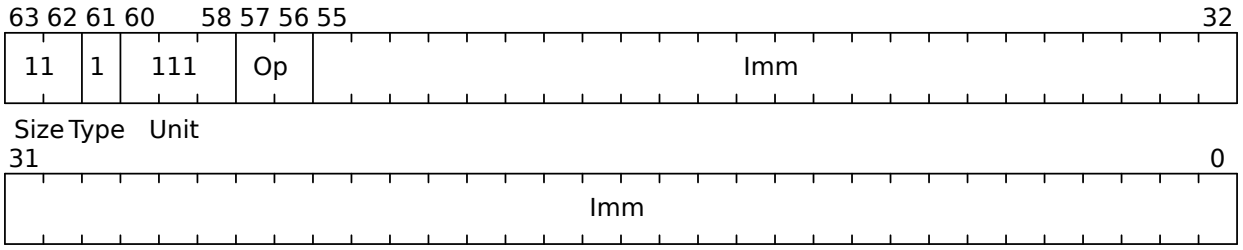
64S - 64-bit Standard Format



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)[12]
		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[1]
		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[1]
		3 - JI[1]

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

[1] 64-bit format only

[2] 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 de_badDetect.v

```
1 //Raisin64 Decode Unit - Bad Opcode Detector
2
3 module de_badDetect(
4     //# {{data|Instruction Data}}
5     input[63:0] opcodeIn,
6
7     //# {{control|Control Signals}}
8     output badOpcode
9     );
```

(continues on next page)

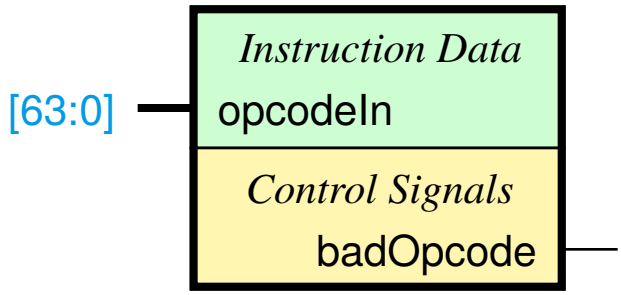


Fig. 1: de_badDetect.v

(continued from previous page)

```
10
11      `include "de_isa_def.vh"
12
13      reg badOpcode_pre;
14      wire is16, is32, is64;
15
16      assign badOpcode = badOpcode_pre;
17
18      assign is16 = ~opcodeIn[63];
19      assign is32 = opcodeIn[63:62] == 2'h2;
20      assign is64 = opcodeIn[63:62] == 2'h3;
21
22      //Detects and flags invalid opcodes
23      always @(*)
24      begin
25          badOpcode_pre = 0;
26
27          if(is16 && opcodeIn[62:60] == 3'h7) badOpcode_pre = 1;
28          else if(is32 | is64)
29          begin
30              case(opcodeIn[61:56])
31                  `OP_BAD_02, `OP_BAD_03, `OP_BAD_0B, `OP_BAD_14, `OP_BAD_15,
32                  `OP_BAD_16, `OP_BAD_17, `OP_BAD_18, `OP_BAD_19, `OP_BAD_1A,
33                  `OP_BAD_1B, `OP_BAD_22, `OP_BAD_23, `OP_BAD_2B: badOpcode_pre = 1;
34                  `OP_F*, `OP_LUI, `OP_JALI, `OP_JI: if(is32) badOpcode_pre = 1;
35              endcase
36          end
37      end
38  endmodule
```

5.2.2 de_canonicalize.v

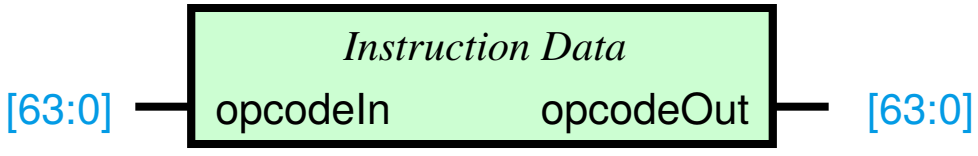


Fig. 2: de_canonicalize.v

```
//Raisin64 Decode Unit - Opcode Canonicalization
//Converts compact instructions into their true 64-bit native format

module de_canonicalize(
    //# {{data|Instruction Data}}
    input[63:0] opcodeIn,
    output[63:0] opcodeOut
    );

    `include "de_isa_def.vh"

    reg[63:0] opcodeOut_pre;
    assign opcodeOut = opcodeOut_pre;

    //Expands input instruction into full 64-bit format
    always @(*)
    begin
        opcodeOut_pre = 64'h0;

        //Set the output size field appropriately
        opcodeOut_pre[63:62] = 2'h3;

        //16-Bit input instructions
        if(~opcodeIn[63])
        begin
            //Switch on Opcode field
            case(opcodeIn[62:60])
                `OP16_ADD:      opcodeOut_pre[61:56] = `OP_ADD; //ADD Rd = Rd + Rs
                `OP16_SUB:      opcodeOut_pre[61:56] = `OP_SUB; //SUB Rd = Rd - Rs
                `OP16_ADDI:     opcodeOut_pre[61:56] = `OP_ADDI; //ADDI Rd = Rd + sign_
↪extend(imm)
                `OP16_SUBI:     opcodeOut_pre[61:56] = `OP_SUBI; //SUBI Rd = Rd - imm
                `OP16_SYSCALL: opcodeOut_pre[61:56] = `OP_SYSCALL; //SYSCALL
                `OP16_J:        opcodeOut_pre[61:56] = `OP_J; //J Rs
                `OP16_JAL:      opcodeOut_pre[61:56] = `OP_JAL; //JAL Rs
            endcase

            opcodeOut_pre[55:50] = opcodeIn[59:54]; //Put the Rd/Rs1 into Rd
            opcodeOut_pre[49:44] = 6'h0; //Rd2 is not used in this format
            opcodeOut_pre[43:38] = opcodeIn[59:54]; //Put the Rd/Rs1 into Rs1
            opcodeOut_pre[37:32] = opcodeIn[53:48]; //Populate Rs2 (imm type␣
↪instructions ignore this)

            //Sign extended immediate field and populate
            opcodeOut_pre[31:0]  = {{26{opcodeIn[53]}},opcodeIn[53:48]}; //reg type␣
↪instructions ignore this
        end

        //32-bit instructions
        else if(~opcodeIn[62])
        begin
            opcodeOut_pre[61:56] = opcodeIn[61:56]; //Set Type/Unit/Op fields
            opcodeOut_pre[55:50] = opcodeIn[55:50]; //Set Rd

            //32I-Type instruction
            if(opcodeIn[61]) begin
                opcodeOut_pre[43:38] = opcodeIn[49:44]; //Set Rs1
```

(continues on next page)

(continued from previous page)

```
55
56                  //Sign-Extended type
57                  if(opcodeIn[60] || //Most sign-extended Ops
58                     opcodeIn[60:58]==3'h0 || //ADDI/SUBI
59                     opcodeIn[61:56]==`OP_SLTI ||
60                     opcodeIn[61:56]==`OP_SGTI)
61                        opcodeOut_pre[31:0] = {{20{opcodeIn[43]}},opcodeIn[43:32]};
62                  //Non Sign-extended type
63                  else opcodeOut_pre[11:0] = opcodeIn[43:32];
64
65              //32R-Type instructions
66              end else begin
67                  //Set the other operands
68                  opcodeOut_pre[49:32] = opcodeIn[49:32];
69
70                  //No immediate
71              end
72          end
73          //64-bit instructions
74          else opcodeOut_pre = opcodeIn;
75      end
76  endmodule
```

5.2.3 de_isa_def.vh

```
1   //Defines related to the ISA
2
3   //16-bit Opcodes
4   `define OP16_ADD     3'h0
5   `define OP16_SUB     3'h1
6   `define OP16_ADDI    3'h2
7   `define OP16_SUBI    3'h3
8   `define OP16_SYSCALL 3'h4
9   `define OP16_J       3'h5
10  `define OP16_JAL     3'h6
11  `define OP16_ADD     3'h7
12
13  //32 and 64-bit Opcodes
14  `define OP_ADD            6'h00
15  `define OP_SUB            6'h01
16  `define OP_BAD_02         6'h02
17  `define OP_BAD_03         6'h03
18  `define OP_SLT            6'h04
19  `define OP_SLTU           6'h05
20  `define OP_SGT            6'h06
21  `define OP_SGTU           6'h07
22  `define OP_SLL            6'h08
23  `define OP_SRA            6'h09
24  `define OP_SRL            6'h0a
25  `define OP_BAD_0B         6'h0b
26  `define OP_AND            6'h0c
27  `define OP_NOR            6'h0d
28  `define OP_OR             6'h0e
29  `define OP_XOR            6'h0f
30  `define OP_MUL            6'h10
```

(continues on next page)

(continued from previous page)

```
`define OP_MULU          6'h11
`define OP_DIV           6'h12
`define OP_DIVU          6'h13
`define OP_BAD_14        6'h14
`define OP_BAD_15        6'h15
`define OP_BAD_16        6'h16
`define OP_BAD_17        6'h17
`define OP_BAD_18        6'h18
`define OP_BAD_19        6'h19
`define OP_BAD_1A        6'h1a
`define OP_BAD_1B        6'h1b
`define OP_SYSCALL       6'h1c
`define OP_F*            6'h1d
`define OP_JAL           6'h1e
`define OP_J             6'h1f
`define OP_ADDI          6'h20
`define OP_SUBI          6'h21
`define OP_BAD_22        6'h22
`define OP_BAD_23        6'h23
`define OP_SLTI          6'h24
`define OP_SLTIU         6'h25
`define OP_SGTI          6'h26
`define OP_SGTIU         6'h27
`define OP_SLLI          6'h28
`define OP_SRAI          6'h29
`define OP_SRLI          6'h2a
`define OP_BAD_2B        6'h2b
`define OP_ANDI          6'h2c
`define OP_NORI          6'h2d
`define OP_ORI           6'h2e
`define OP_XORI          6'h2f
`define OP_LW            6'h30
`define OP_L32           6'h31
`define OP_L16           6'h32
`define OP_L8            6'h33
`define OP_LUI           6'h34
`define OP_L32S          6'h35
`define OP_L16S          6'h36
`define OP_L8S           6'h37
`define OP_SW            6'h38
`define OP_S32           6'h39
`define OP_S16           6'h3a
`define OP_S8            6'h3b
`define OP_BEQ           6'h3c
`define OP_BEQAL         6'h3d
`define OP_JALI          6'h3e
`define OP_JI            6'h3f
```

5.2.4 ex_advint.v

```
//Raisin64 Execute Unit - Advanced Integer Unit

//For now, we just use the intrinsic * and / operations.  These do have the
//advantage of mapping to DSP hard-blocks on our example SoC design.

```

(continues on next page)

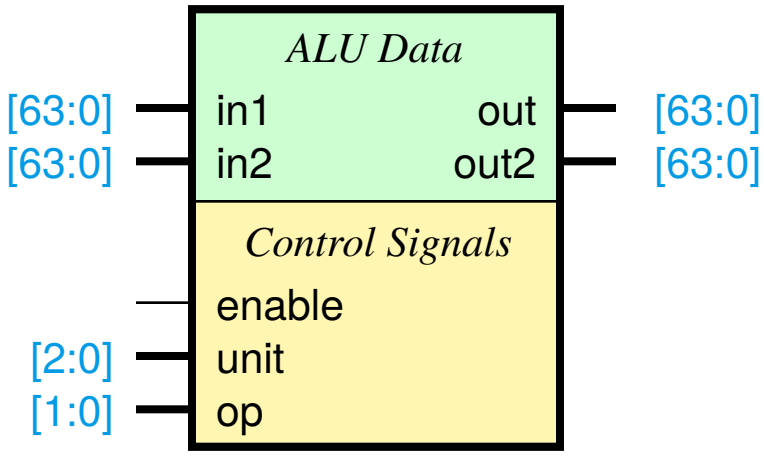


Fig. 3: ex_advint.v

(continued from previous page)

```
6  module ex_advint(
7      //# {{data|ALU Data}}
8      input[63:0] in1,
9      input[63:0] in2,
10     output[63:0] out,
11     output[63:0] out2,
12
13     //# {{control|Control Signals}}
14     input enable,
15     input[2:0] unit,
16     input[1:0] op
17     );
18
19     reg[127:0] out_pre;
20     assign out = out_pre[63:0];
21     assign out2 = out_pre[127:64];
22
23     always @(*)
24     begin
25         out_pre = 128'h0;
26
27         if(enable & unit==3'h4) begin
28             case(op)
29                 0: out_pre = $signed(in1) * $signed(in2); //MUL
30                 1: out_pre = in1 * in2; //MULU
31                 2: begin //DIV
32                     out_pre[63:0] = $signed(in1) / $signed(in2);
33                     out_pre[127:64] = $signed(in1) % $signed(in2);
34                    end
35                 3: begin //DIVU
36                     out_pre[63:0] = in1 / in2;
37                     out_pre[127:64] = in1 % in2;
38                    end
39             endcase
40         end
41     end
42 endmodule
```

5.2.5 ex_alu.v

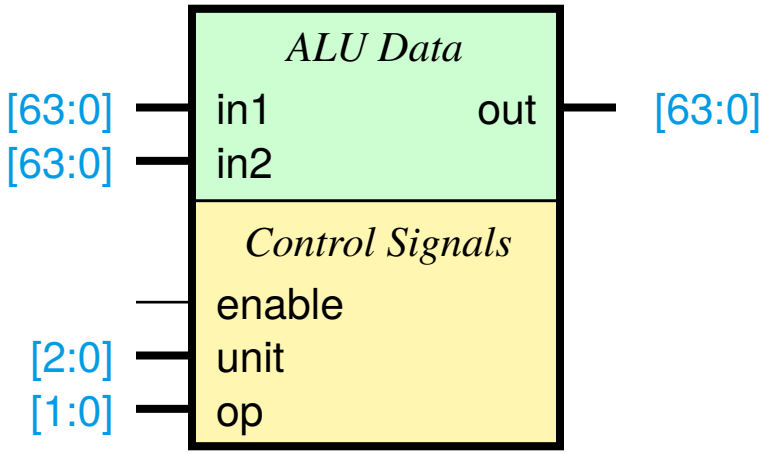


Fig. 4: ex_alu.v

```
//Raisin64 Execute Unit - Integer ALU

module ex_alu(
    //# {{data|ALU Data}}
    input[63:0] in1,
    input[63:0] in2,
    output[63:0] out,

    //# {{control|Control Signals}}
    input enable,
    input[2:0] unit,
    input[1:0] op
    );

    reg[63:0] out_pre;
    assign out = out_pre;

    always @(*)
    begin
        out_pre = 64'h0;
        if(enable) case(unit)
            3'h0: //Basic integer math
                case(op[0])
                    0,2: out_pre = in1 + in2; //ADD
                    1,3: out_pre = in1 - in2; //SUB
                endcase
            3'h1: //Compare/Set
                case(op)
                    0: out_pre = $signed(in1) < $signed(in2) ? 64'h1 : 64'h0; //SLT
                    1: out_pre = in1 < in2 ? 64'h1 : 64'h0; //SLTU
                    2: out_pre = $signed(in1) > $signed(in2) ? 64'h1 : 64'h0; //SGT
                    3: out_pre = in1 > in2 ? 64'h1 : 64'h0; //SGTU
                endcase
            3'h2: //Shift
                case(op)
                    0: out_pre = in1 << {|in2[63:6], in2[5:0]}; //SLL
```

(continues on next page)

(continued from previous page)

```
                    1: out_pre = in1 >>> {|in2[63:6], in2[5:0]}; //SRA
                    2,3: out_pre = in1 >> {|in2[63:6], in2[5:0]}; //SRL
                endcase
            3'h3: //Bitwise Operations
                case(op)
                    0: out_pre = in1 & in2; //AND
                    1: out_pre = in1 ~| in2; //NOR
                    2: out_pre = in1 | in2; //OR
                    3: out_pre = in1 ^ in2; //XOR
                endcase
        endcase
    end
endmodule
```

5.2.6 Raisin64.v

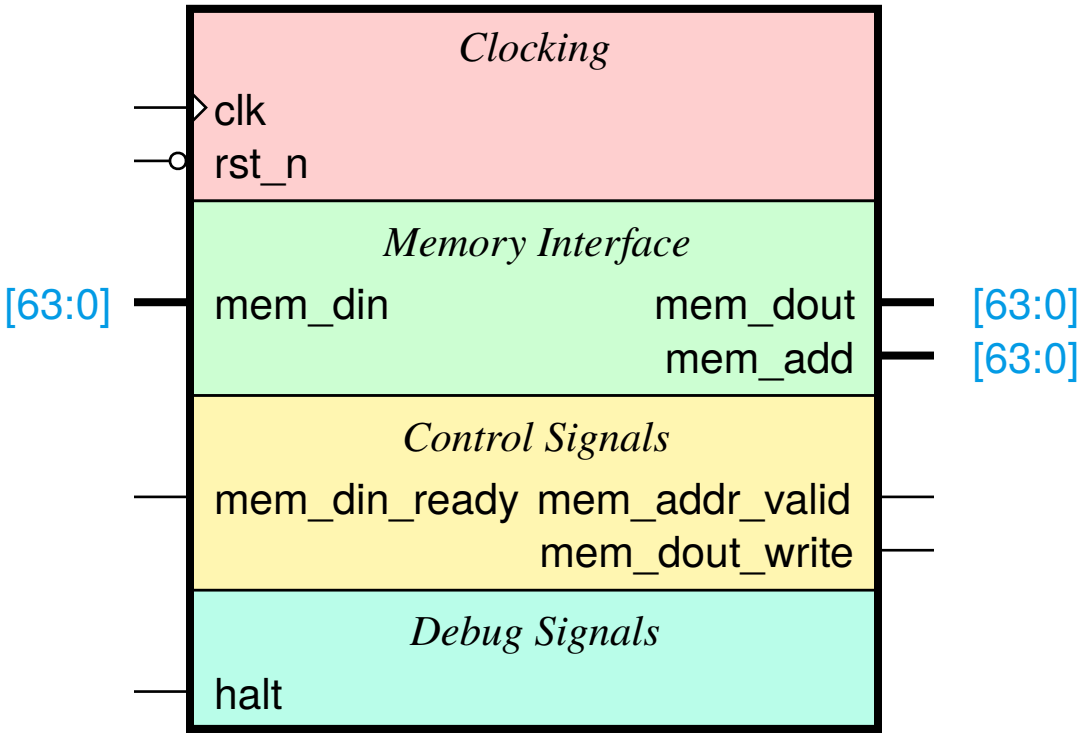


Fig. 5: Raisin64.v

```
/*
 * Raisin64 CPU
 */

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

```

(continues on next page)

(continued from previous page)

```
    //# {{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
```

5.2.7 rf_reg.v

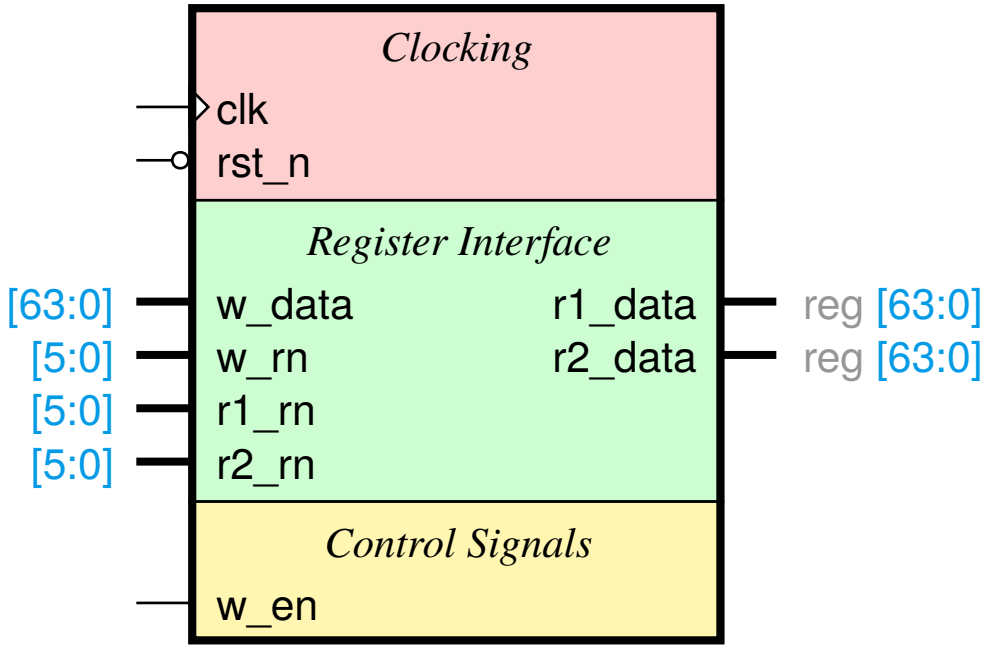


Fig. 6: rf_reg.v

```
//Raisin64 Register File
//Registered dual-ported register file two read, one write port

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

    //# {{data|Register Interface}}
    input[63:0] w_data,
    input[5:0] w_rn,
```

(continues on next page)

(continued from previous page)

```
    output reg[63:0] r1_data,
    output reg[63:0] r2_data,
    input[5:0] r1_rn,
    input[5:0] r2_rn,

    //# {{control|Control Signals}}
    input w_en

    //# {{debug|Debug Signals}}
    );

    reg[63:0] file[1:63];
    integer i;

    //TODO if place-route isn't smart enough to figure it out and use block ram, have two register files with write ports cross connected so we get "two" read ports on a clock cycle using multiple simple dual-port rams.
    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            for(i = 1; i<64; i = i+1) file[i] <= 64'h0;
        end else begin
            if(w_en && w_rn!=6'h0) file[w_rn] <= w_data;
        end
    end

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) r1_data <= 64'h0;
        else if(r1_rn==6'h0) r1_data <= 64'h0;
        else r1_data <= file[r1_rn];
    end

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) r2_data <= 64'h0;
        else if(r2_rn==6'h0) r2_data <= 64'h0;
        else r2_data <= file[r2_rn];
    end
endmodule
```