
Raisin64 Documentation

Release 0.1

Christopher Parish

Dec 11, 2018

CONTENTS

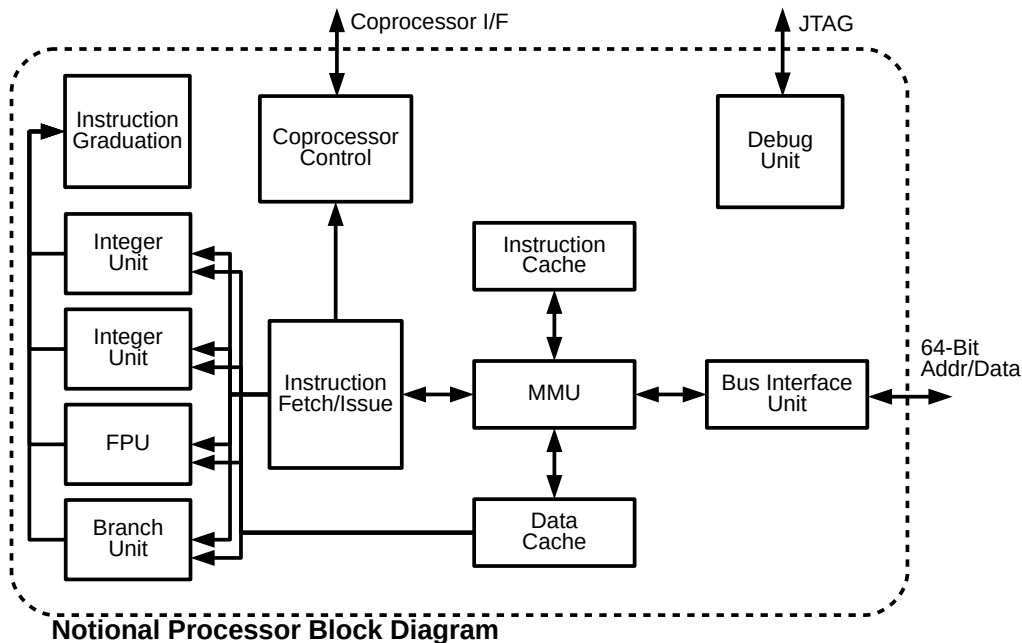
1	Raisin64 CPU	3
1.1	Overview	4
1.2	Pipeline Stages	4
1.2.1	Fetch Unit	4
1.2.2	Decode Unit	4
1.2.3	Register File	4
1.2.4	Schedule Unit	4
1.2.5	Execution Units	4
1.2.5.1	Integer Unit	4
1.2.5.2	Advanced Integer Unit	4
1.2.5.3	Branch Unit	4
1.2.5.4	Memory Unit	4
1.2.6	Commit Unit	4
1.3	Debug Unit	4
1.4	Proposed Extensions	4
1.4.1	MMU	4
1.4.2	Interrupt Unit	6
1.4.3	Caches	6
1.4.4	References	6
2	Code Snippets and Software	7
2.1	Handling Interrupts	7
2.2	Initializing the MMU	7
3	Tools	9
3.1	Assembler	9
3.2	Debugging	9
3.2.1	Getting OpenOCD	9
4	Nexys 4 DDR Reference Implementation	11
4.1	SoC Peripherals	11
4.2	Required Hardware	11
4.3	Synthesizing the Core	11
5	Reference Index	13
5.1	Raisin64 Instruction Set	13
5.1.1	Overview	13
5.1.2	Instruction Format	14
5.1.3	16-bit formats	14
5.1.3.1	16R - 16-bit Register Format	14

5.1.3.2	16I - 16-bit Immediate Format	14
5.1.3.3	16-bit OpCode Table	15
5.1.4	32/64-bit Formats	15
5.1.4.1	32R - 32-bit Register Format	15
5.1.4.2	32I - 32-bit Immediate Format	15
5.1.4.3	64S - 64-bit Standard Format	15
5.1.4.4	64J - 64-bit Jump Format	16
5.1.4.5	32 and 64-bit Unit/Op Table	16
5.1.5	Instructions	18
5.1.5.1	ADD - Integer Add	18
5.1.5.2	ADDI - Integer Add Immediate	19
5.1.5.3	AND - Bitwise AND	20
5.1.5.4	ANDI - Bitwise AND Immediate	21
5.1.5.5	BEQ - Branch if Equal	22
5.1.5.6	BEQAL - Branch if Equal And Link	23
5.1.5.7	DIV - Integer Divide	24
5.1.5.8	DIVU - Unsigned Integer Divide	25
5.1.5.9	F* - FPU Call	26
5.1.5.10	J - Jump	27
5.1.5.11	JAL - Jump and Link	28
5.1.5.12	JALI - Jump and Link Immediate	29
5.1.5.13	Jl - Jump Immediate	30
5.1.5.14	L16 - Load 16-bit	31
5.1.5.15	L16S - Load 16-bit Sign-Extend	32
5.1.5.16	L32 - Load 32-bit	33
5.1.5.17	L32S - Load 32-bit Sign-Extend	34
5.1.5.18	L8 - Load 8-bit	35
5.1.5.19	L8S - Load 8-bit Sign-Extend	36
5.1.5.20	LUI - Load Upper Immediate	37
5.1.5.21	LW - Load 64-bit Word	38
5.1.5.22	MUL - Integer Multiply	39
5.1.5.23	MULU - Unsigned Integer Multiply	40
5.1.5.24	NOR - Bitwise NOR	41
5.1.5.25	NORI - Bitwise NOR Immediate	42
5.1.5.26	OR - Bitwise OR	43
5.1.5.27	ORI - Bitwise OR Immediate	44
5.1.5.28	S16 - Store 16-bit	45
5.1.5.29	S32 - Store 32-bit	46
5.1.5.30	S8 - Store 8-bit	47
5.1.5.31	SGT - Set 1 if Greater Than	48
5.1.5.32	SGTI - Set 1 if Greater Than Immediate	49
5.1.5.33	SGTIU - Set 1 if Greater Than Immediate Unsigned	50
5.1.5.34	SGTU - Set 1 if Greater Than Unsigned	51
5.1.5.35	SLL - Shift Left Logical	52
5.1.5.36	SLLI - Shift Left Logical Immediate	53
5.1.5.37	SLT - Set 1 if Less Than	54
5.1.5.38	SLTI - Set 1 if Less Than Immediate	55
5.1.5.39	SLTIU - Set 1 if Less Than Immediate Unsigned	56
5.1.5.40	SLTU - Set 1 if Less Than Unsigned	57
5.1.5.41	SRA - Shift Right Arithmetic	58
5.1.5.42	SRAI - Shift Right Arithmetic Immediate	59
5.1.5.43	SRL - Shift Right Logical	60
5.1.5.44	SRLI - Shift Right Logical Immediate	61
5.1.5.45	SUB - Integer Subtract	62

5.1.5.46	SUBI - Integer Subtract Immediate	63
5.1.5.47	SW - Store 64-bit Word	64
5.1.5.48	SYSCALL - System Call	65
5.1.5.49	XOR - Bitwise XOR	66
5.1.5.50	XORI - Bitwise XOR Immediate	67
5.2	Verilog Module Index	68
5.2.1	de_badDetect.v	68
5.2.2	de_canonicalize.v	69
5.2.3	de_isa_def.vh	70
5.2.4	ex_advint.v	70
5.2.5	ex_alu.v	72
5.2.6	Raisin64.v	76
5.2.7	rf_reg.v	79
6	Footnotes	81

Raisin64 (*RISC Architecture with In-order Superscalar INterlocked-pipeline*) is a pure 64-bit CPU design created as part of an educational project. Inspired by the architecture of the [MIPS R10000](#) and [POWER3](#), Raisin64 employs multiple specialized pipelines for integer operations, branching, load/store, etc presently using a simplified issue system appropriate for the scope of a semester-long project.

Unlike most superscalar designs, Raisin64 does not re-order instructions or use register renaming¹ but instead provides a larger architectural register file of 63x64-bit registers.



Major features of the Raisin64 include:

Bits 64-bit

Instructions 50 Opcodes (with 16, 32, and 64-bit formats)

Design RISC (Harvard Architecture²)

Type Register-Register

Branching Compare and Branch

Endianness Big

Registers 63 (R0 = 0)

¹ Raisin64 will issue instructions out-of-order assuming subsequent instructions are dependency-free and the appropriate execution unit is available.

² Split-Cache Modified Harvard when *proposed caches and MMU* are introduced.

RAISIN64 CPU

- *Overview*
- *Pipeline Stages*
 - *Fetch Unit*
 - *Decode Unit*
 - *Register File*
 - *Schedule Unit*
 - *Execution Units*
 - * *Integer Unit*
 - * *Advanced Integer Unit*
 - * *Branch Unit*
 - * *Memory Unit*
 - *Commit Unit*
- *Debug Unit*
- *Proposed Extensions*
 - *MMU*
 - *Interrupt Unit*
 - *Caches*
 - *References*

1.1 Overview

1.2 Pipeline Stages

1.2.1 Fetch Unit

1.2.2 Decode Unit

1.2.3 Register File

1.2.4 Schedule Unit

1.2.5 Execution Units

1.2.5.1 Integer Unit

1.2.5.2 Advanced Integer Unit

1.2.5.3 Branch Unit

1.2.5.4 Memory Unit

1.2.6 Commit Unit

1.3 Debug Unit

1.4 Proposed Extensions

Future Work

While out-of-scope for the present period of the project, some initial development was done on *Caches*, an *MMU*, and *Interrupt Unit*, primarily to ensure that they can be integrated into the design without significant modification to the processing pipeline.

These extensions will make the processor capable of running a general purpose operating system (such as Linux) without resorting to software emulation of customarily present hardware.

1.4.1 MMU

Nearly all general purpose operating systems depend on a *Memory Management Unit* to provide the virtual addressing used by userspace processes¹². The MMU presents each process with an illusory linear address space potentially overlapping with many other processes. Along with the *Translation Lookaside Buffer*, an MMU critically allows processes to be placed at arbitrary physical addresses (wherever the RAM happens to be free), with pages of that physical memory mapped at whatever virtual addresses the process expects.

¹ <https://www.kernel.org/doc/Documentation/nommu-mmmap.txt>

² <https://wiki.netbsd.org/projects/project/mmu-less/>

In the Raisin64, the MMU also acts as the first point where the instruction and data caches have a unified window into physical memory, making the processor a split-cache Harvard architecture. Beyond the [page tables](#) which are conventionally placed in main memory, the MMU control registers will be present in the machine's memory-map and be accessible in a kernel-mode un-mapped region (that is, the memory addresses used to access the registers will never be mapped by the MMU and will always be passed through without translation).

Proposed MMU Specs:

Page Size 16KB Fixed

VA Width 47-Bits sign-extended

Page Table Three Level (3x 11-bit entries and 15-bit offset)

The virtual addressing scheme takes inspiration from several modern processor designs as a way to constrain the number of legal virtual addresses while not inhibiting the physical address space available to the MMU. While the virtual addresses are 64-bits, bits 63:47 must be sign-extended (i.e. replicated) from bit 46. This breaks the address space into several proposed regions:

Address	Purpose
0xFFFFFFFF_FFFFFFFF - 0xFFFC000_00000000	Kernel-Mode Mapped
0xFFFBFFFF_FFFFFFFF - 0xFFFF8000_00000000	Kernel-Mode Unmapped
0xFFFF7FFF_FFFFFFFF - 0x00008000_00000000	Invalid
0x00007FFF_FFFFFFFF - 0x00000000_00000000	User-Mode Mapped

The following figure from ARM on the MIPS processor's memory map conveys the general principle of using the kernel-mode unmapped segment to allow access to IO registers (MMU configuration included) which are present at a fixed physical address:

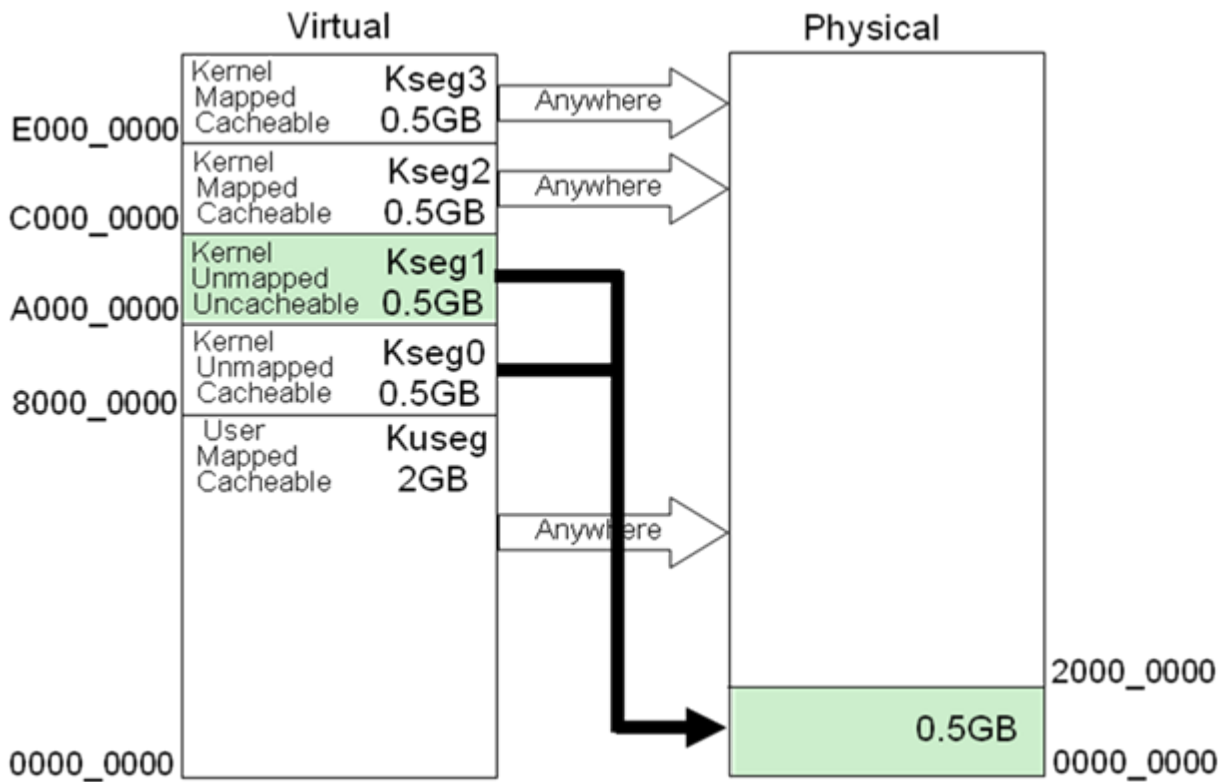


Fig. 1: From ARM AN235 Section 3.4³

1.4.2 Interrupt Unit

An Interrupt/Exception unit will be necessary to properly implement virtual memory. Attempting to access an unmapped, evicted, or privileged page from a userspace process should cause the operating system to take over and mitigate the situation (either by loading the page or terminating the process).

The Raisin64's processing pipeline will need some modifications to the *Commit Unit* although first steps have already been taken to add a mechanism allowing register and memory writes to be deferred and re-ordered. This can be expanded with program counter tracking information to ensure that the precise location of an interrupt can be recovered and the processor did not commit the pending results of an issued instruction later in the (now aborted) instruction stream.

1.4.3 Caches

Relatively simple compared to the MMU or Interrupt Unit, caches will likely have the largest impact on performance of the processor. As the processing pipeline uses a Harvard architecture, the first level of caching is made up of a separate Instruction and Data cache. Each will sit on their respective data ports and provide a small number of highly/fully associative entries that are *virtually indexed and virtually tagged*.

This scheme will necessitate the flushing of the cache on a context switch, but as the only known implementations of the Raisin64 are on FPGAs (without the benefit of hardware content-addressable memory), the caches need to be small regardless and flushing their content on a context-switch will affect only a small number of entries.

Proposed Cache Specs:

L1 Cache Split Instruction/Data

L1 Data Small N-Way/Fully Associative

L1 Instruction Small N-Way/Fully Associative

L1 Tag Scheme Virtually Indexed, Virtually Tagged

L2 Cache Large Unified 2-Way Set Associative

L2 Tag Scheme Physically Indexed, Physically Tagged

While a second level cache between the MMU and main memory may be advantageous, the (comparatively) slow clock rates available from an FPGA but with full speed hardware-accelerated RAM access may eliminate any benefit of another cache.

1.4.4 References

³ http://infocenter.arm.com/help/topic/com.arm.doc.dai0235c/index.html#arm_toc13

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

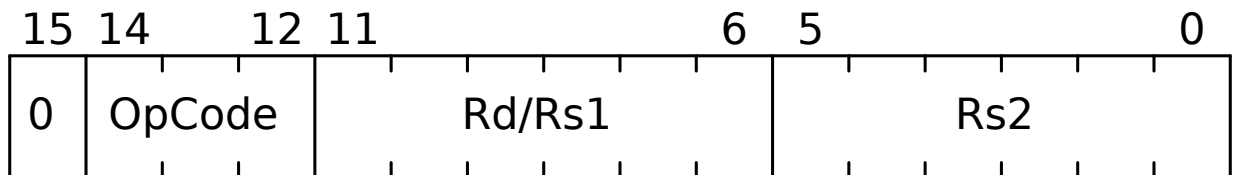
There are 6 instruction formats in Raisin64, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

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

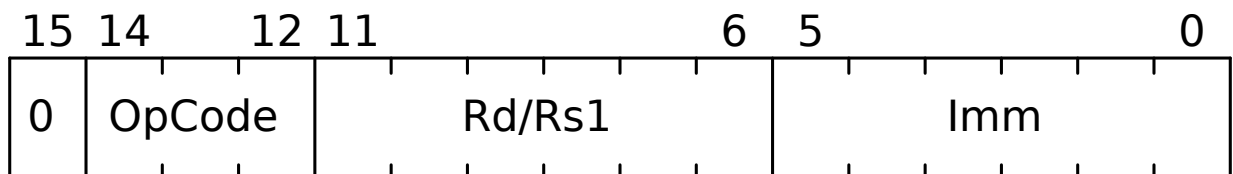
5.1.3.1 16R - 16-bit Register Format



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. $Rd = Rd + Rs$).

5.1.3.2 16I - 16-bit Immediate Format



Size

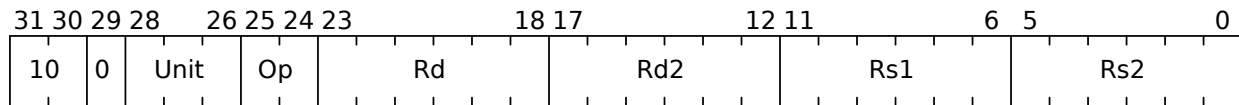
The 16-bit immediate format is used only for ADDI and SUBI, allowing for small increment and decrement operations in a compact format.

5.1.3.3 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

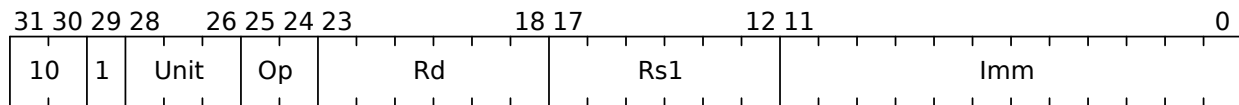
5.1.4.1 32R - 32-bit Register Format



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.

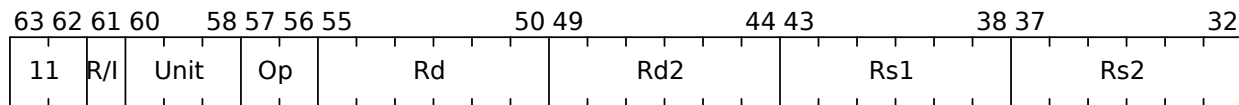
5.1.4.2 32I - 32-bit Immediate Format



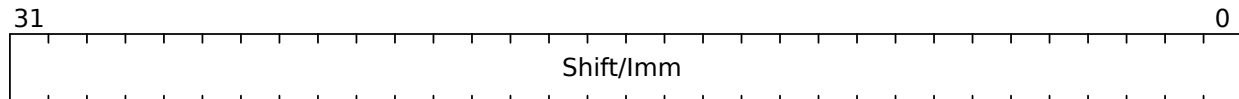
Size Type

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.

5.1.4.3 64S - 64-bit Standard Format



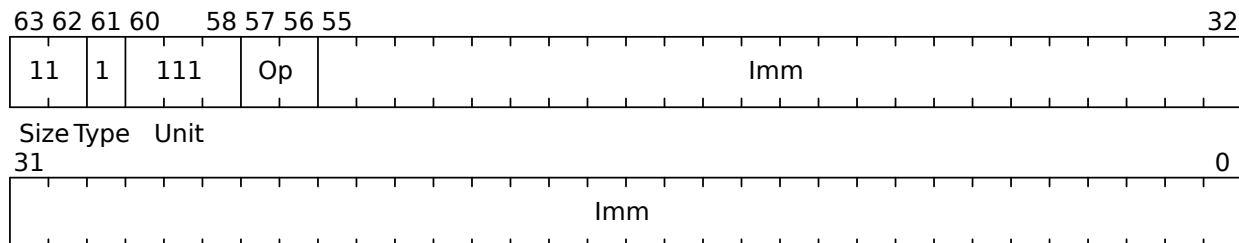
Size



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).

5.1.4.4 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.

5.1.4.5 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 - J
		3 - JAL
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 - JI ¹
		3 - JALI ¹

¹ 64-bit format only² 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.

5.1.5 Instructions

5.1.5.1 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

5.1.5.2 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

5.1.5.3 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

5.1.5.4 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

5.1.5.5 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

5.1.5.6 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

5.1.5.7 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

5.1.5.8 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

5.1.5.9 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

5.1.5.10 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 2

16-bit Opcode 0x5

32-bit Opcode 0x9E

64-bit Opcode 0xDE

5.1.5.11 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 3

16-bit Opcode 0x6

32-bit Opcode 0x9F

64-bit Opcode 0xDF

5.1.5.12 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 3

16-bit Opcode NONE

32-bit Opcode NONE

64-bit Opcode 0xFF

5.1.5.13 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 2

16-bit Opcode NONE

32-bit Opcode NONE

64-bit Opcode 0xFE

5.1.5.14 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

5.1.5.15 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

5.1.5.16 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

5.1.5.17 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

5.1.5.18 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

5.1.5.19 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

5.1.5.20 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

5.1.5.21 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

5.1.5.22 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

5.1.5.23 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

5.1.5.24 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

5.1.5.25 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

5.1.5.26 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

5.1.5.27 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

5.1.5.28 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

5.1.5.29 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

5.1.5.30 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

5.1.5.31 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

5.1.5.32 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

5.1.5.33 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

5.1.5.34 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

5.1.5.35 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

5.1.5.36 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

5.1.5.37 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

5.1.5.38 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

5.1.5.39 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

5.1.5.40 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

5.1.5.41 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

5.1.5.42 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

5.1.5.43 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

5.1.5.44 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

5.1.5.45 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

5.1.5.46 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

5.1.5.47 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

5.1.5.48 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

5.1.5.49 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

5.1.5.50 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

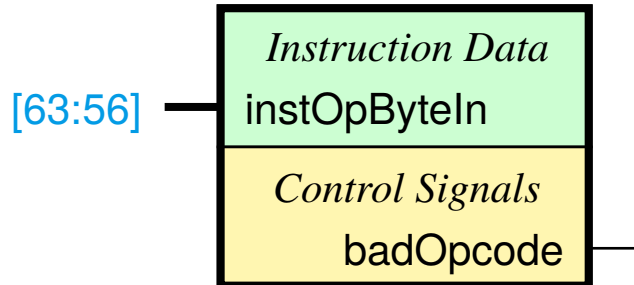


Fig. 1: de_badDetect.v

```

1  //Raisin64 Decode Unit - Bad Opcode Detector
2
3  module de_badDetect (
4      //# {{data/Instruction Data}}
5      input [63:56] instOpByteIn,
6
7      //# {{control/Control Signals}}
8      output badOpcode
9  );
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 = ~instOpByteIn[63];
19  assign is32 = instOpByteIn[63:62] == 2'h2;
20  assign is64 = instOpByteIn[63:62] == 2'h3;
21
22  //Detects and flags invalid opcodes
23  always @(*)
24  begin
25      badOpcode_pre = 0;
26
27      if(is16 && instOpByteIn[62:60] == 3'h7) badOpcode_pre = 1;
28      else if(is32 | is64)
29      begin
30          case(instOpByteIn[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_FSTAR, `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

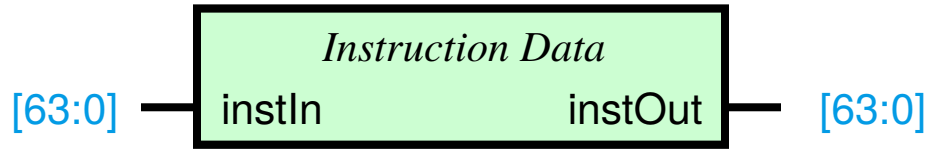


Fig. 2: de_canonicalize.v

```

1  //Raisin64 Decode Unit - Opcode Canonicalization
2  //Converts compact instructions into their true 64-bit native format
3
4  module de_canonicalize(
5      ## {{data/Instruction Data}}
6      input [63:0] instIn,
7      output [63:0] instOut
8  );
9
10     `include "de_isa_def.vh"
11
12     reg [63:0] instOut_pre;
13     assign instOut = instOut_pre;
14
15     //Expands input instruction into full 64-bit format
16     always @(*)
17     begin
18         instOut_pre = 64'h0;
19
20         //Set the output size field appropriately
21         instOut_pre[63:62] = 2'h3;
22
23         //16-Bit input instructions
24         if (~instIn[63])
25         begin
26             //Switch on Opcode field
27             case (instIn[62:60])
28                 `OP16_ADD:    instOut_pre[61:56] = `OP_ADD; //ADD Rd = Rd + Rs
29                 `OP16_SUB:    instOut_pre[61:56] = `OP_SUB; //SUB Rd = Rd - Rs
30                 `OP16_ADDI:   instOut_pre[61:56] = `OP_ADDI; //ADDI Rd = Rd + sign_
31                 ↪extend(imm)
32                 `OP16_SUBI:    instOut_pre[61:56] = `OP_SUBI; //SUBI Rd = Rd - imm
33                 `OP16_SYSCALL: instOut_pre[61:56] = `OP_SYSCALL; //SYSCALL
34                 `OP16_J:      instOut_pre[61:56] = `OP_J; //J Rs
35                 `OP16_JAL:    instOut_pre[61:56] = `OP_JAL; //JAL Rs
36             endcase
37
38             instOut_pre[55:50] = instIn[59:54]; //Put the Rd/Rs1 into Rd
39             instOut_pre[49:44] = 6'h0; //Rd2 is not used in this format
40             instOut_pre[43:38] = instIn[59:54]; //Put the Rd/Rs1 into Rs1
41             instOut_pre[37:32] = instIn[53:48]; //Populate Rs2 (imm type instructions_
42             ↪ignore this)
43
44             //Sign extended immediate field and populate
45             instOut_pre[31:0] = {{26{instIn[53]}}, instIn[53:48]}; //reg type_
46             ↪instructions ignore this

```

(continues on next page)

(continued from previous page)

```

44     end
45
46     //32-bit instructions
47     else if(~instIn[62])
48     begin
49         instOut_pre[61:56] = instIn[61:56]; //Set Type/Unit/Op fields
50         instOut_pre[55:50] = instIn[55:50]; //Set Rd
51
52         //32I-Type instruction
53         if(instIn[61]) begin
54             instOut_pre[43:38] = instIn[49:44]; //Set Rs1
55
56             //Sign-Extended type
57             if(instIn[60] || //Most sign-extended Ops (remember, JALI and JI are
↳invalid for 32I)
58                 instIn[60:58]==3'h0 || //ADDI/SUBI signed versions sign extend
59                 instIn[61:56]==`OP_SLTI || //SLTI and SGTI are signed and sign
↳extend
60                 instIn[61:56]==`OP_SGTI)
61                 instOut_pre[31:0] = {{20{instIn[43]}},instIn[43:32]};
62             //Non Sign-extended type
63             else instOut_pre[11:0] = instIn[43:32];
64
65             //32R-Type instructions
66         end else begin
67             //Set the other operands
68             instOut_pre[49:32] = instIn[49:32];
69
70             //No immediate
71         end
72     end
73     //64-bit instructions
74     else instOut_pre = instIn;
75 end
76 endmodule

```

5.2.3 de_isa_def.vh

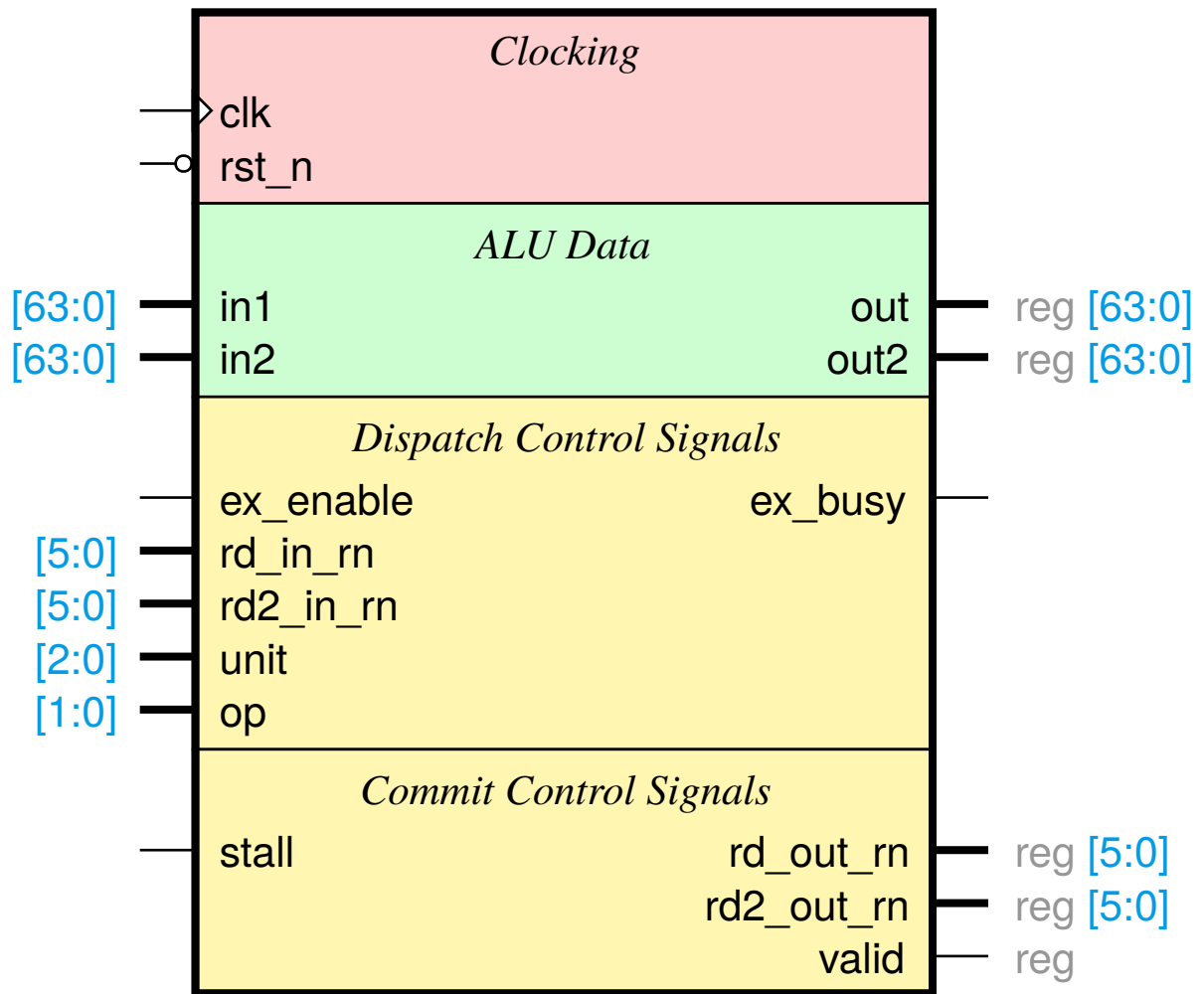
5.2.4 ex_advint.v

```

1  //Raisin64 Execute Unit - Advanced Integer Unit
2
3  module ex_advint(
4      //# {{clocks|Clocking}}
5      input clk,
6      input rst_n,
7
8      //# {{data|ALU Data}}
9      input [63:0] in1,
10     input [63:0] in2,
11     output reg [63:0] out,
12     output reg [63:0] out2,
13
14     //# {{control|Dispatch Control Signals}}

```

(continues on next page)

Fig. 3: `ex_advint.v`

(continued from previous page)

```

15     input ex_enable,
16     output ex_busy,
17     input[5:0] rd_in_rn,
18     input[5:0] rd2_in_rn,
19     input[2:0] unit,
20     input[1:0] op,
21
22     //# {{control/Commit Control Signals}}
23     output reg[5:0] rd_out_rn,
24     output reg[5:0] rd2_out_rn,
25     output reg valid,
26     input stall
27 );
28
29 wire[63:0] out_pre;
30 wire[63:0] out2_pre;
31
32 //We allow the next result to register when we aren't explicitly beign
33 //stalled by the next stage (i.e. our result has somewhere to go).
34 always @(posedge clk or negedge rst_n)
35 begin
36     if(~rst_n) begin
37         valid <= 0;
38         out <= 64'h0;
39         out2 <= 64'h0;
40         rd_out_rn <= 6'h0;
41         rd2_out_rn <= 6'h0;
42     end else begin
43         valid <= ex_enable;
44         out <= out_pre;
45         out2 <= out2_pre;
46         rd_out_rn <= rd_in_rn;
47         rd2_out_rn <= rd2_in_rn;
48     end
49 end
50
51 initial begin
52     if(stall&ex_enable) $error("Told to execute AdvInt when commit was stalled");
53 end
54
55 //As this is a one-cycle stage for now, busy is simple
56 assign ex_busy = stall;
57
58 ex_advint_s1 ex_advint_s1_1(
59     .in1(in1), .in2(in2), .out(out_pre), .out2(out2_pre),
60     .enable(ex_enable), .unit(unit), .op(op)
61 );
62
63 endmodule

```

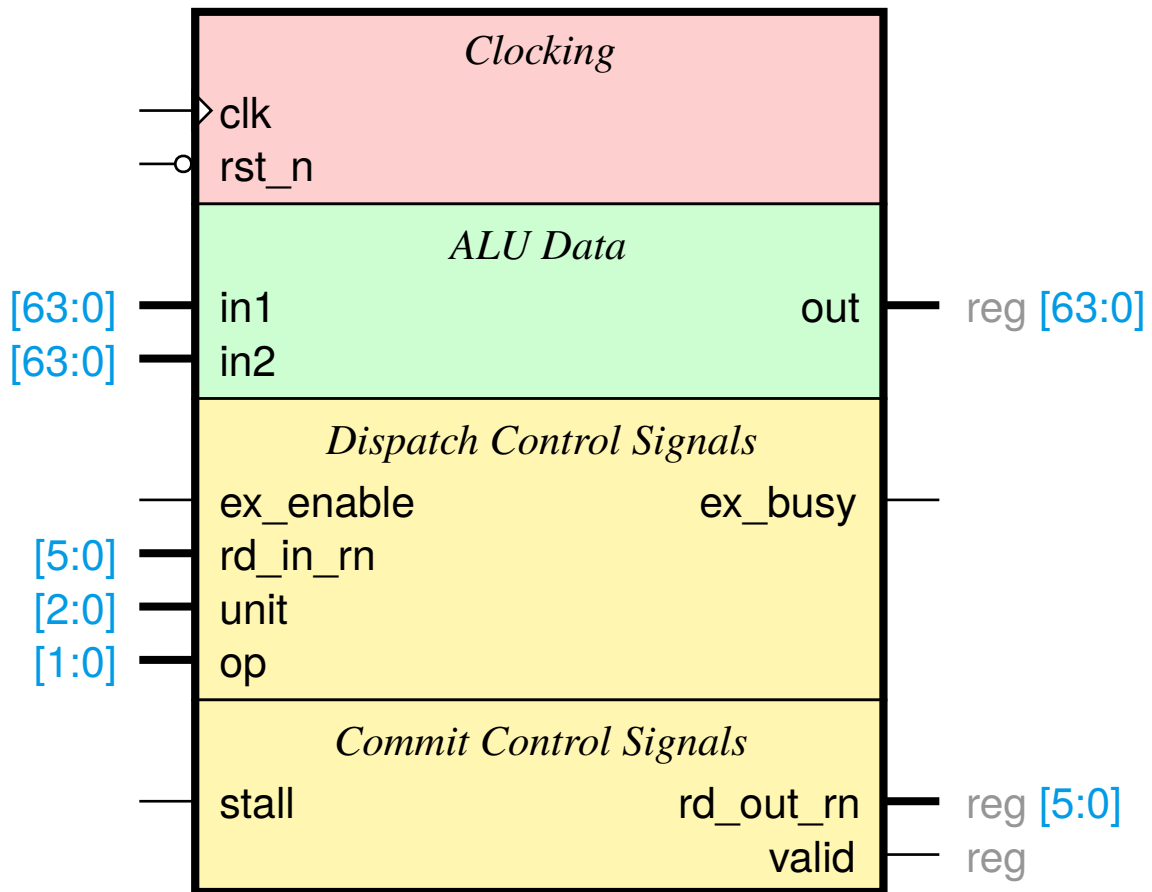
5.2.5 ex_alu.v

```

1 //Raisin64 Execute Unit - Integer ALU
2
3 module ex_alu(

```

(continues on next page)


Fig. 4: `ex_alu.v`

(continued from previous page)

```

4    /// {{clocks|Clocking}}
5    input clk,
6    input rst_n,
7
8    /// {{data|ALU Data}}
9    input [63:0] in1,
10   input [63:0] in2,
11   output reg [63:0] out,
12
13   /// {{control|Dispatch Control Signals}}
14   input ex_enable,
15   output ex_busy,
16   input [5:0] rd_in_rn,
17   input [2:0] unit,
18   input [1:0] op,
19
20   /// {{control|Commit Control Signals}}
21   output reg [5:0] rd_out_rn,
22   output reg valid,
23   input stall
24   );
25
26   wire [63:0] out_pre;
27
28   //We allow the next result to register when we aren't explicitly beign
29   //stalled by the next stage (i.e. our result has somewhere to go).
30   always @(posedge clk or negedge rst_n)
31   begin
32       if (~rst_n) begin
33           valid <= 0;
34           out <= 64'h0;
35           rd_out_rn <= 6'h0;
36       end else if (~stall) begin
37           valid <= ex_enable;
38           out <= out_pre;
39           rd_out_rn <= rd_in_rn;
40       end
41   end
42
43   initial begin
44       if (stall & ex_enable) $error("Told to execute ALU when commit was stalled");
45   end
46
47   //As this is a one-cycle stage, busy is simple
48   assign ex_busy = stall;
49
50   ex_alu_sl ex_alu_sl_1(
51       .in1(in1), .in2(in2), .out(out_pre), .enable(ex_enable),
52       .unit(unit), .op(op)
53   );
54
55 endmodule

```

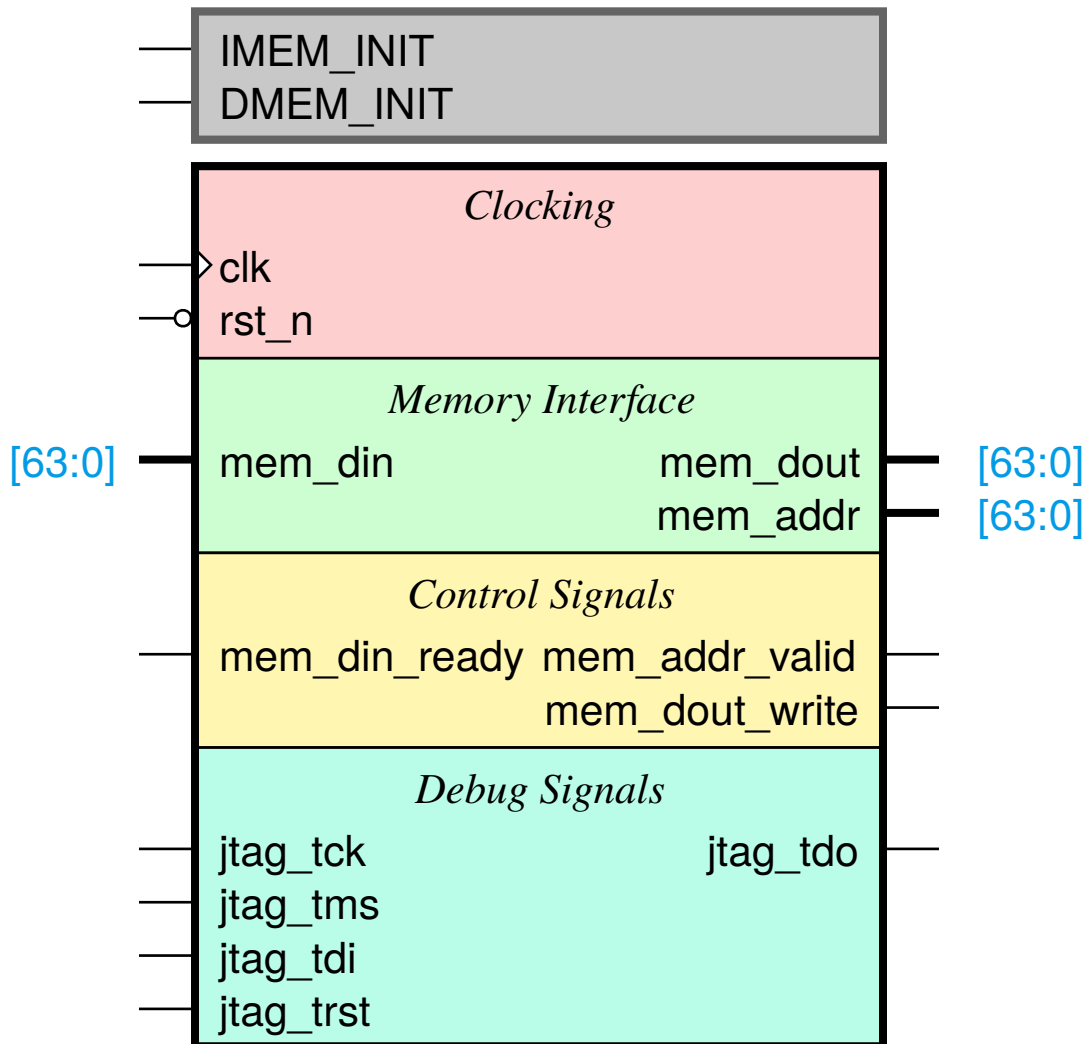



Fig. 5: Raisin64.v

5.2.6 Raisin64.v

```

1  /*
2   * Raisin64 CPU
3   */
4
5  module raisin64 (
6      // # {{clocks|Clocking}}
7      input clk,
8      input rst_n,
9
10     // # {{data|Memory Interface}}
11     input [63:0] mem_din,
12     output [63:0] mem_dout,
13     output [63:0] mem_addr,
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 jtag_tck,
22     input jtag_tms,
23     input jtag_tdi,
24     input jtag_trst,
25     output jtag_tdo
26 );
27
28 parameter IMEM_INIT = "";
29 parameter DMEM_INIT = "";
30
31 // // Debug Signals // //
32 wire dbg_resetrn_cpu, dbg_halt_cpu;
33 wire cpu_rst_n;
34 assign cpu_rst_n = rst_n & dbg_resetrn_cpu;
35
36 wire [63:0] dbg_imem_addr;
37 wire [63:0] dbg_imem_to_ram;
38 wire dbg_imem_ce;
39 wire dbg_imem_we;
40
41 wire [63:0] dbg_dmem_addr;
42 wire [63:0] dbg_dmem_to_ram;
43 wire dbg_dmem_ce;
44 wire dbg_dmem_we;
45
46
47 // // Instruction RAM // //
48 wire [63:0] effective_imem_addr;
49 wire [63:0] effective_imem_data_to_cpu;
50
51 reg imem_data_ready;
52 wire imem_addr_valid;
53 wire [63:0] imem_addr;
54 wire [63:0] imem_data;
55

```

(continues on next page)

(continued from previous page)

```

56  assign effective_imem_addr      = dbg_halt_cpu ? dbg_imem_addr : imem_addr;
57  assign effective_imem_data_to_cpu = dbg_halt_cpu ? 64'h0 : imem_data;
58
59  always @(posedge clk or negedge rst_n) begin
60      if(~rst_n) imem_data_ready <= 0;
61      else imem_data_ready <= imem_addr_valid;
62  end
63
64  ram #(
65      .NUM_BYTES(2*1024),
66      .INIT_FILE(IMEM_INIT)
67  ) imem (
68      .clk(clk),
69      .we(dbg_imem_we), .cs(1'b1),
70      .write_width(2'h0),
71      .addr(effective_imem_addr),
72      .data_in(dbg_imem_to_ram),
73      .data_out(imem_data)
74  );
75
76
77  //////////// Data RAM ////////////
78  wire[63:0] effective_dmem_addr;
79  wire[63:0] effective_dmem_to_ram;
80
81  wire[63:0] dmem_addr;
82  wire[63:0] dmem_to_ram;
83  wire[63:0] dmem_to_cpu;
84  wire[63:0] dmem_from_ram;
85  wire[1:0] dmem_write_width;
86  reg dmem_cycle_complete;
87  wire dmem_rstrobe;
88  wire dmem_wstrobe;
89
90  wire io_space;
91
92  assign effective_dmem_addr      = dbg_halt_cpu ? dbg_dmem_addr : dmem_addr;
93  assign effective_dmem_to_ram    = dbg_halt_cpu ? dbg_dmem_to_ram : dmem_to_ram;
94
95  //TODO For now, the external memory bus is just for data memory. When the time
96  //comes for caches, this will change to the unified external memory bus.
97  assign dmem_to_cpu              = io_space ? mem_din : dmem_from_ram;
98  assign mem_dout                 = effective_dmem_to_ram;
99  assign mem_addr                 = effective_dmem_addr;
100 assign mem_addr_valid            = 1;
101 assign mem_dout_write            = dbg_halt_cpu ? dbg_dmem_we : dmem_wstrobe;
102
103  //Because the memory interface will change dramatically in the next revision,
104  ↪there
105  //is no reason to create special logic to handle misaligned accesses into data
106  ↪space
107  //in case an IO unit requires it (the ram modules handle this condition
108  ↪internally).
109  //Instead we simply state misaligned IO access it is unsupported (for now).
110  always @(*) begin
111      if((dmem_rstrobe|dmem_wstrobe) & io_space & |dmem_write_width & ~clk) begin
112          $display("Unaligned data IO access not supported in this revision");
113      end
114  end

```

(continues on next page)

(continued from previous page)

```

110         $finish;
111     end
112 end
113
114 memory_map memory_map_internal(
115     .addr(mem_addr),
116     .io(io_space)
117 );
118
119 always @(posedge clk or negedge cpu_rst_n)
120 begin
121     if(~cpu_rst_n) dmem_cycle_complete <= 0;
122     else if(io_space & mem_din_ready) dmem_cycle_complete <= 1;
123     else if(dmem_rstrobe) dmem_cycle_complete <= 1;
124     else if(dmem_wstrobe) dmem_cycle_complete <= 1;
125     else dmem_cycle_complete <= 0;
126 end
127
128 ram #(
129     .NUM_BYTES(512),
130     .INIT_FILE(DMEM_INIT)
131 ) dmem (
132     .clk(clk),
133     .we(~io_space & (dmem_wstrobe|dbg_dmem_we)), .cs(~io_space & (dmem_
134 ↪wstrobe|dmem_rstrobe|dbg_dmem_ce)),
135     .write_width(dmem_write_width),
136     .addr(effective_dmem_addr),
137     .data_in(effective_dmem_to_ram),
138     .data_out(dmem_from_ram)
139 );
140
141 /////////////// Raisin64 Execution Core ///////////////
142 pipeline pipeline1(
143     .clk(clk),
144     .rst_n(cpu_rst_n),
145     .imem_addr(imem_addr),
146     .imem_data(effective_imem_data_to_cpu),
147     .imem_data_valid(imem_data_ready),
148     .imem_addr_valid(imem_addr_valid),
149     .dmem_addr(dmem_addr), .dmem_dout(dmem_to_ram),
150     .dmem_din(dmem_to_cpu),
151     .dmem_cycle_complete(dmem_cycle_complete & ~dmem_rstrobe & ~dmem_wstrobe),
152     .dmem_write_width(dmem_write_width),
153     .dmem_rstrobe(dmem_rstrobe),
154     .dmem_wstrobe(dmem_wstrobe)
155 );
156
157 /////////////// JTAG Module ///////////////
158 debug_control debug_if(
159     .jtag_tck(jtag_tck),
160     .jtag_tms(jtag_tms),
161     .jtag_tdo(jtag_tdo),
162     .jtag_tdi(jtag_tdi),
163     .jtag_trst(jtag_trst),
164     .cpu_clk(clk),

```

(continues on next page)

(continued from previous page)

```
166     .sys_rstn(rst_n),
167     .cpu_imem_addr(dbg_imem_addr),
168     .cpu_debug_to_imem_data(dbg_imem_to_ram),
169     .cpu_imem_to_debug_data(imem_data),
170     .cpu_imem_we(dbg_imem_we),
171     .cpu_imem_ce(dbg_imem_ce),
172     .cpu_dmem_addr(dbg_dmem_addr),
173     .cpu_debug_to_dmem_data(dbg_dmem_to_ram),
174     .cpu_imem_to_debug_data_ready(dbg_imem_ce & ~dbg_imem_we),
175     .cpu_dmem_to_debug_data_ready(dbg_dmem_ce & ~dbg_dmem_we),
176     .cpu_dmem_to_debug_data(dmem_to_cpu),
177     .cpu_dmem_we(dbg_dmem_we),
178     .cpu_dmem_ce(dbg_dmem_ce),
179     .cpu_resetrn_cpu(dbg_resetrn_cpu),
180     .cpu_halt_cpu(dbg_halt_cpu)
181 );
182
183 endmodule
```

5.2.7 rf_reg.v

FOOTNOTES