Raisin64 Documentation

Release 0.1

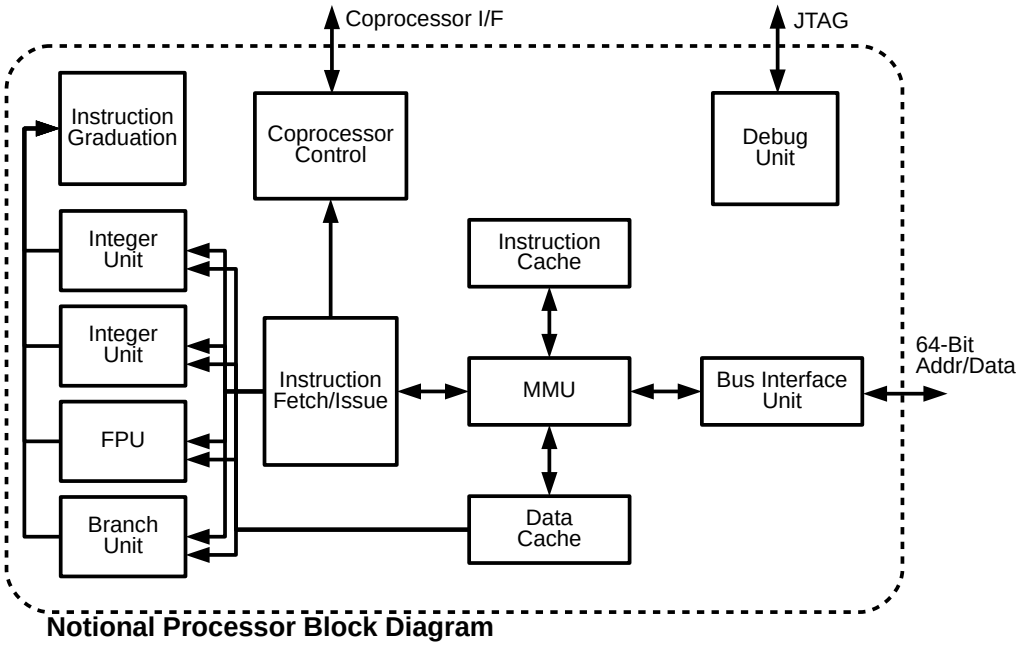
Christopher Parish

Dec 12, 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. 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 renaming[1] but instead provides a larger architectural register file of 63x64-bit registers.



Notional Processor Block Diagram

Major features of the Raisin64 include:

Bits 64-bit

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

Design RISC (Harvard Architecture[2])

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 dependancy-free and the appropriate execution unit is available.

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

CHAPTER

ONE

RAISIN64 CPU

1.1 Overview

The Raisin64 CPU

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 processes[1][2]. 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.

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 - 0xFFFFC000_00000000	Kernel-Mode Mapped
0xFFFFBFFF_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:

1.4.2 Interrupt Unit

An Interrupt/Exception unit will be necessary to properly implement virtual memory. Attempting to access an unmapped, evicted, or privilaged 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] https://www.kernel.org/doc/Documentation/nommu-mmap.txt
[2] https://wiki.netbsd.org/projects/project/mmu-less/
[3] http://infocenter.arm.com/help/topic/com.arm.doc.dai0235c/index.html#arm_toc13

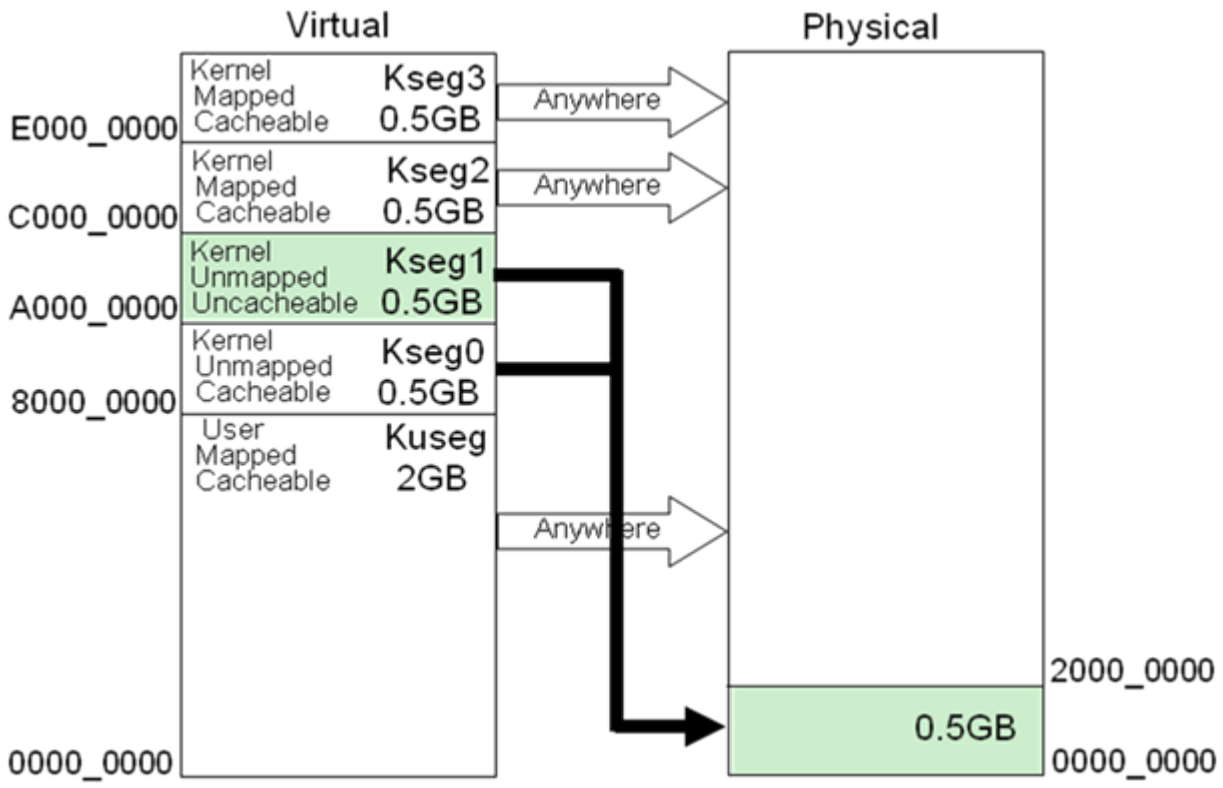


Fig. 1: From ARM AN235 Section 3.4[3]

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

CHAPTER

TWO

CODE SNIPPETS AND SOFTWARE

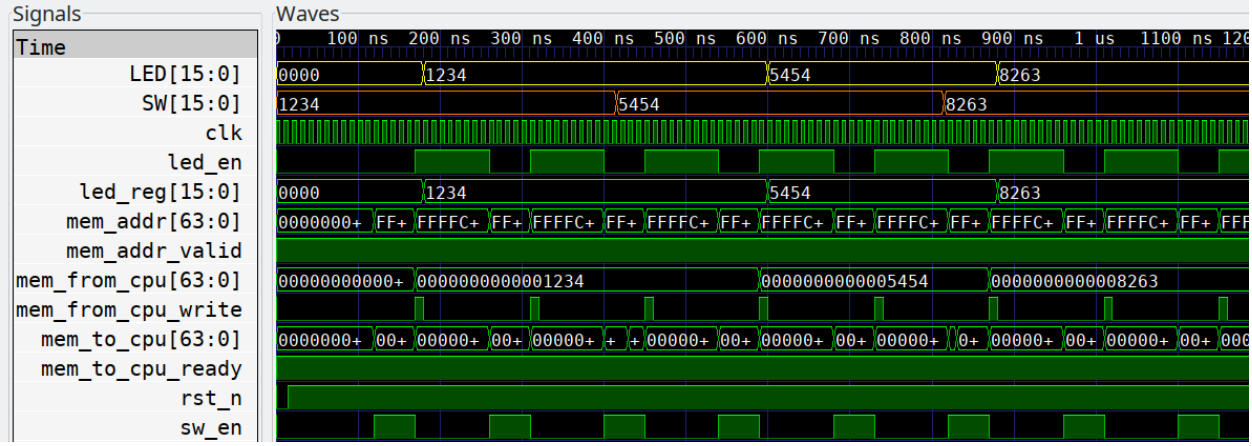
Each instruction page contains an example of how to use that specific opcode which will not be repeated here. Instead, a few simple programs will pre presented that work with the *Nexys 4 DDR Reference Implementation* demonstrating aspects of the ISA or use of the hardware.

While having a completely different opcode format and compact instruction support, the Raisin64 drew inspiration from MIPS for it's instruction set and mnemonics. As a result, several programs I created for a previous academic MIPS design were easily ported to the Raisin64.

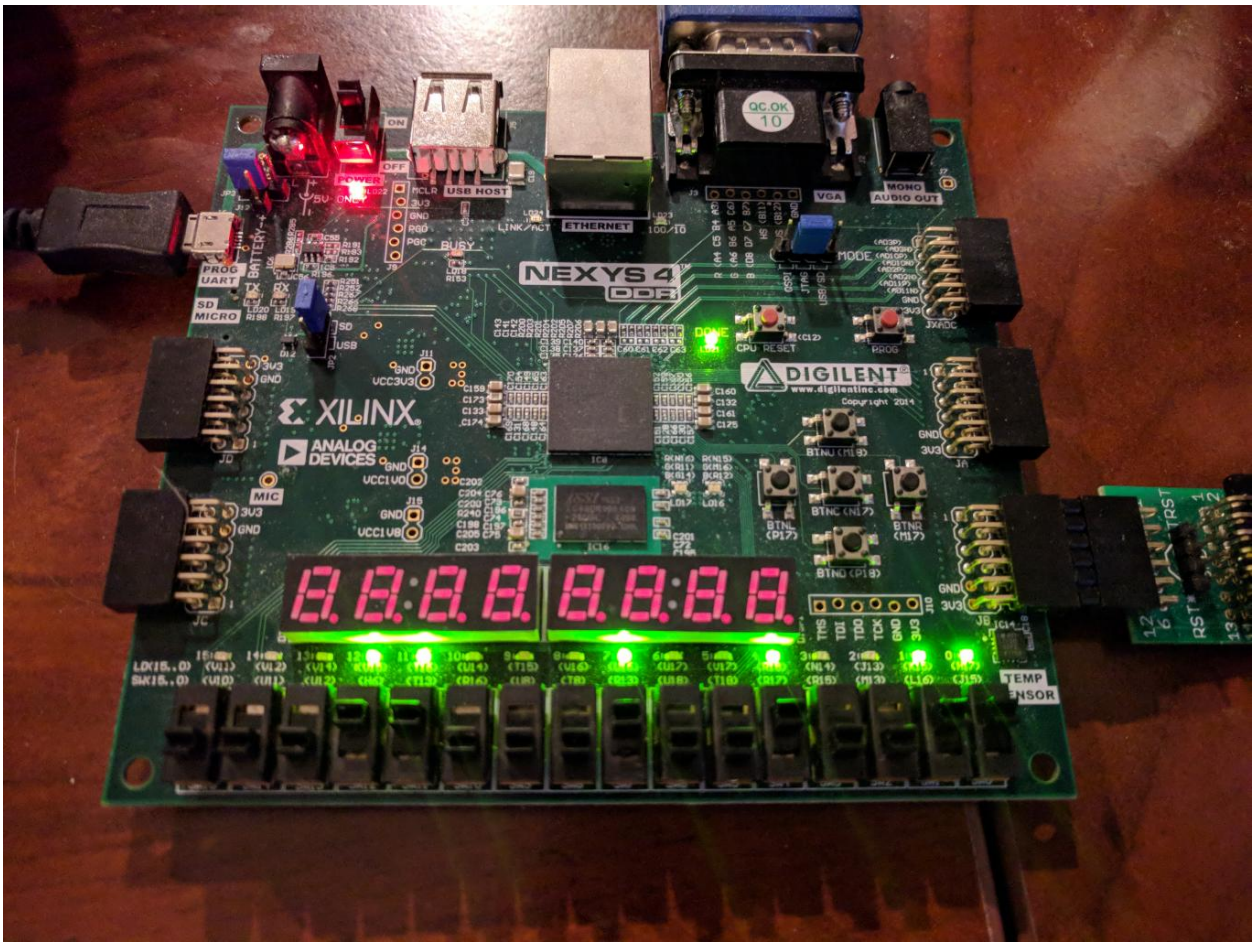
2.1 Switch to LED

The Switch to LED program is the simplest proof of life for the Nexys 4 DDR board, reading the present position of the switches, and mirroring them onto the array of LEDs located immediately above them.

```
.set SW_LADR, 0x00008000
.set LED_LADR, 0x00004000

.text
#Load the sign-extended upper portion of the IO space in R1
lui $r1, 0xFFFFC000

sw_loop:
ori $r2, $r1, SW_LADR  #Load the switch address in R2
lw $r3, ($r2)          #and read into R3
ori $r2, $r1, LED_LADR #Now load LED address into R2
sw $r3, ($r2)          #And store R3 into *R2
ji sw_loop             #Repeat

add $r0, $r0, $r0      #NOP (not a delay slot) TODO Fix for assembler frag␣
↪misalignment
```



From a simulation of the external hardware, the LED port can be seen tracking the SW (switch) port soon after it changes. This can be run on the real hardware with the same result:



2.2 VGA Demo Program

As a non-trivial test of the processor, a demo program was created exercising the VGA subsystem of the Nexys 4 DDR board located at 0xFFFFC000_0004xxxx on the data memory bus. In addition to the switch to LED functionality

above, it draws a hello world string and continuously iterates through the character set and color options on the lower half of the display.

The assembly demonstrates the use of a stack, as well as useful GNU assembler tools like defines, macros, data labels, alignment, etc. Also available at: https://github.com/ChrisPVille/raisin64-nexys4ddr/blob/master/software/demo.S

```
#-------------------------------------------------------------------------------
#Macros and defines to make life easier

.set IO_HADR,  0xFFFFC000
.set SW_LADR,  0x00008000
.set LED_LADR, 0x00004000
.set VGA_LADR, 0x00040000

.set COLOR_W,  0xF
.set COLOR_R,  0xC
.set COLOR_G,  0xA
.set COLOR_B,  0x9
.set COLOR_Y,  0xE

.set COL, 240
.set ROW, 68

#Loads the character and calls printChar (increments R16; R18 needs to be set)
.macro printCharImm char
    addi $r17, $zero, \char
    jali printChar
    addi $r16, $r16, 1
.endm

.macro friendly_print col, row, attrib_byte, str_ptr
    addi $r16, $zero, \col
    addi $r17, $zero, \row
    addi $r18, $zero, \attrib_byte
    addi $r19, $zero, \str_ptr
    jali printStr
.endm

.macro fn_enter
    addi  $sp, $sp, -8 #Allocate 1 word on the stack
    sw    $lr, ($sp) #Store the current lr on the stack
.endm

.macro fn_exit
    lw   $lr, ($sp) #Restore the original lr
    addi $sp, $sp, 8 #Free the stack space we used
    j    $lr #Return
.endm

#-------------------------------------------------------------------------------
#Data segment (for the data RAM)
.data

#Stack space (grows down towards zero)
stack: .space 8*8
stack_init_head:

#String storage
```

(continues on next page)

(continued from previous page)

```
hello_str: .asciz "Hello, World!"
greet_str: .asciz "Greetings from "

.align 9 #Fill 512

#-------------------------------------------------------------------------------
#Text segment (for the instruction ROM/RAM)
.text

reset:
    #Setup the stack
    addi $sp, $zero, stack_init_head

    #Load the sign-extended upper portion of the IO space in R1
    lui  $r1, IO_HADR
    ori  $r2, $zero, 0xFFFF
    ori  $r3, $r1, LED_LADR  #Now load LED address into R3
    sw   $r2, ($r3)          #And store R2 into *R3

    #Clear the display
    jali clearDisp

    #Write the plain strings
    friendly_print 115 20 0x0f hello_str
    friendly_print 110 21 0x0f greet_str

    #Write the colorful Raisin64
    addi $r16, $zero, (21*COL)+125 #Row 21, Col 125
    addi $r18, $zero, COLOR_B
    printCharImm 'R'
    addi $r18, $zero, COLOR_G
    printCharImm 'a'
    addi $r18, $zero, COLOR_Y
    printCharImm 'i'
    addi $r18, $zero, COLOR_R
    printCharImm 's'
    addi $r18, $zero, COLOR_B
    printCharImm 'i'
    addi $r18, $zero, COLOR_G
    printCharImm 'n'
    addi $r18, $zero, COLOR_Y
    printCharImm '6'
    addi $r18, $zero, COLOR_R
    printCharImm '4'

    jali  reset_finloop
    addi  $r5, $zero, COL*ROW #Final character
fin_loop:
    ori   $r4, $r1, SW_LADR   #Load the switch address in R4
    lw    $r3, ($r4)          #and read into R3
    ori   $r4, $r1, LED_LADR  #Now load LED address into R4
    sw    $r3, ($r4)          #And store R3 into *R4
    jali  printChar
    addi  $r17, $r17, 1
    addi  $r18, $r18, 3
    addi  $r16, $r16, 1
    beqal $r16, $r5, reset_finloop
```

(continues on next page)

(continued from previous page)

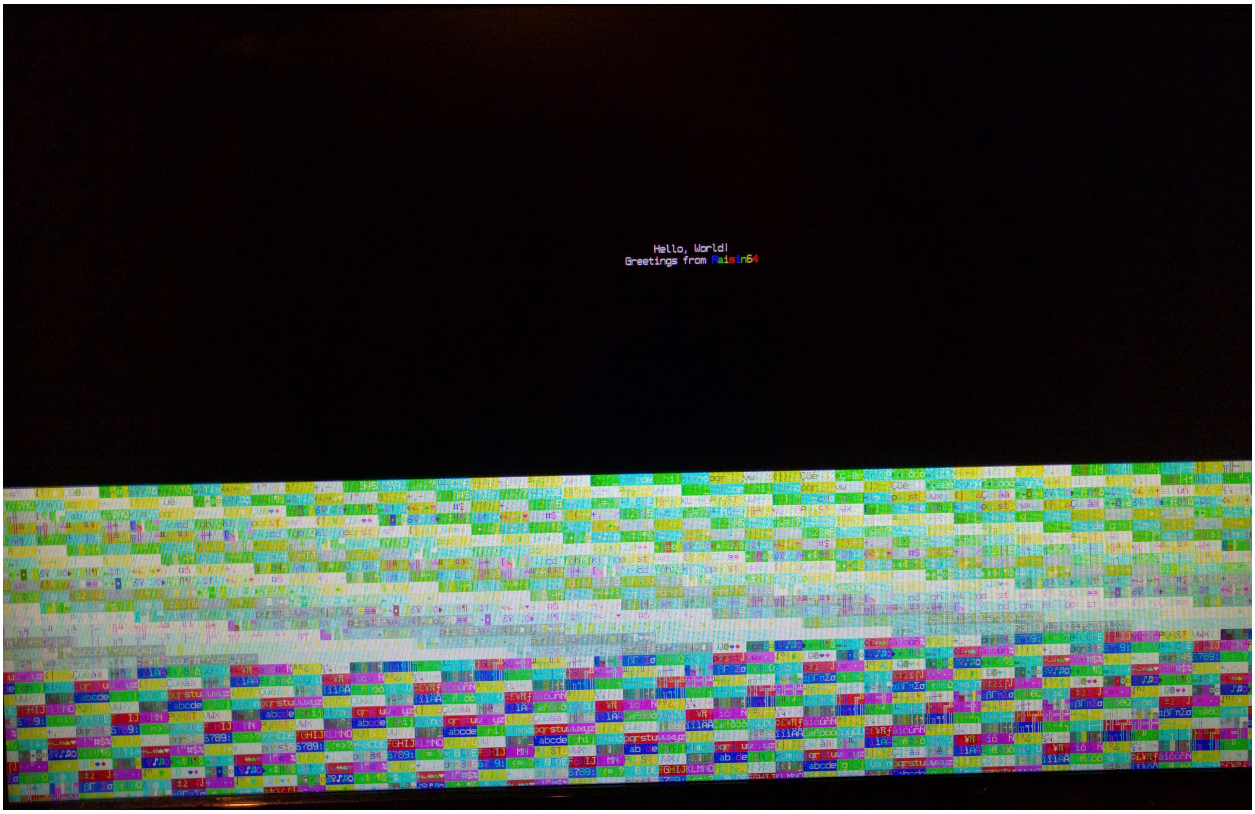
```
    ji    fin_loop            #Repeat
reset_finloop:
    addi  $r16, $zero, COL*40  #Start at row 40
    j     $lr

#Clears display
clearDisp:
    fn_enter
    addi $r16, $zero, ROW*COL
    add  $r17, $zero, $zero
    add  $r18, $zero, $zero
clearDisp_loop:
    beq  $r16, $zero, clearDisp_done
    jali printChar
    subi $r16, $r16, 1
    ji   clearDisp_loop
clearDisp_done:
    fn_exit

#Print ASCII string
# R16: Col
# R17: Row
# R18: Attribute
# R19: ASCII String (reference)
printStr:
    fn_enter
    addi $r4, $zero, COL         #R4 gets Number of Characters in Row
    mul  $r17, $zero, $r17, $r4  #R17 = NumItemsInCol*RowNum
    add  $r16, $r17, $r16        #R16 = Buffer "Character" number

printStr_nextChar:
    l8   $r17, ($r19)    #R17 = Byte in string
    beq  $r17, $zero, printStr_done #Null-Terminator
    jali printChar       #Print the character
    addi $r19, $r19, 1   #Increment pointers
    addi $r16, $r16, 1
    ji   printStr_nextChar
printStr_done:
    fn_exit

#Sends character to video display
# R16: Display Buffer Offset
# R17: ASCII Character
# R18: Packed Attribute
printChar:
    #We are a leaf function (calls no others).
    #Don't bother putting ra on the stack as we
    #won't overwrite it with function calls.
    slli $r20, $r18, 8
    or   $r20, $r20, $r17   #Prepare the packed VGA control word
    andi $r20, $r20, 0xFFFF #and mask it

    #Prepare the base VGA address in R2
    ori  $r2, $r1, VGA_LADR
    slli $r21, $r16, 3  #Shift the buffer "cell" number
    add  $r2, $r21, $r2 #Add the cell number to the address
    sw   $r20, ($r2)    #Store the result
```

(continues on next page)

(continued from previous page)

```
167     j     $lr
168
169 .align 11 #Fill 2K
```

Given the long fill period at reset, the simulation is simultaneously un-interesting and overwhelming. Suffice it to say, it leads to a colorful demo.



CHAPTER

THREE

TOOLS

Being a completely new computer architecture and instruction set, there are were no ready-made tools available for assembly, disassembly, linking, debugging, etc. In the spirit of bootstrapping a new system, it was decided early that if the eventual goal is to run a general purpose operating system on the machine, GCC will be required. GCC leverages binutils, a collection of assembly tools, ELF and object file manipulation utilities, as well as a powerful linker.

Using the preliminary instruction set decided early in the semester, binutils was ported to the Raisin64 ISA while the processor was still being designed. Using an existing target (the moxie as a template, the initial port of binutils was made functional by creating/modifying 26 files across the source.

Future Work

The current Raisin64 GNU Assembler port only constructs the 64-bit version of the instruction set. While the linker, disassembler, and other infrastructure tools should support the smaller instruction words (with some testing done to that effect), the assembler will require significant work outside the scope of the present semester.

While the template architecture was noted for it's reasonable size (architecture definitions and assembler were in the many-hundred-line range instead of the tens-of-thousands range for MIPS and X86), the Raisin64 is quite dissimilar being 64-bit with an entirely different instruction scheme. The actual assembler core was largely rewritten in what became a deep exploration of the binutils architecture.

3.1 Assembler

Being a port of binutils, the Raisin64 assembler should be familiar and comfortable to use for an assembly language programmer supporting the full set of GNU As features. An effort was made to support MIPS-like syntax with `$r0` or `$zero` register numbering

Named Registers:

Number	Name	Purpose
`$r0`	`$zero`	Zero Register
`$r62`	`$sp`	Stack Pointer (no special meaning to processor/convention only)
`$r63`	`$lr`	Link Register (Destination for *JAL*/*BEQAL*/etc.)

The assembler can be invoked as usual for GNU As:

```
raisin64-elf-as <input file> -o <output.elf>
```

which will produce an ELF that can be manipulated with objdump, objcopy, etc.

An example of the assembly process is in assemble.sh which takes an input assembly file, produces the assembled ELF, extracts the `.text` and `.data` sections (containing the instruction and data memories respectively), and converts them from hex to ASCII using the xxd utility. The result is suitable for the `$readmemh` Verilog commands:

```
raisin64-elf-as $1 -o prog.elf &&
raisin64-elf-objdump -s -j .data prog.elf &&
raisin64-elf-objdump -d -j .text prog.elf &&
raisin64-elf-objcopy -O binary -j .text prog.elf imem.bin &&
raisin64-elf-objcopy -O binary -j .data prog.elf dmem.bin &&
xxd -c 8 -ps imem.bin > imem.hex &&
xxd -c 8 -ps dmem.bin > dmem.hex
```

3.1.1 Binary Release

A binary release of the Raisin64 binutils was prepared, compatible with most 64-bit linux systems: https://github.com/ChrisPVille/raisin64-binutils/releases

3.1.2 Building from Source

The Raisin64 port can be obtained here: https://github.com/ChrisPVille/raisin64-binutils.

Binutils is mostly free from external dependencies out of necessity, so it should build without too much drama. Just be sure to configure it for the `raisin64-elf` target. i.e.:

```
./configure --target=raisin64-elf --prefix=<install directory>
make -j<threads>
```

3.2 Debugging

As the Raisin64 was designed with a from-scratch JTAG controller I wrote recently (the JTAGlet), there was no existing support in any tools. Not that JTAG core support would help much given the new ISA, but keeping with the bootstrap theme, a custom configuration script for OpenOCD was created that uses/misuses the scripting interface to provide communication with the processor's JTAG interface, program the memories, and examine the state of the machine.

Future Work

While the scripting interface was a quick way to support my target, the conventional approach is to write support for targets and JTAG controller in C, releasing a new version of OpenOCD (much like I did with binutils). This will be necessary to support remote debugging (via GDB) and will make future development easier.

The configuration script is too large to repeat here, but is accessible at https://github.com/ChrisPVille/raisin64-cpu/blob/master/support/jtag/raisin64_nodeps_openocd.cfg. It is currently configured for a Bus Blaster v3, but can be easily reconfigured for other JTAG probes.

This script is be invoked by the adjacent `programImemDmem.sh <imem.hex> <optional dmem.hex>` or as:

```
openocd -f "raisin64_nodeps_openocd.cfg" -c "init; raisin64_program <imem.hex>
↪<optional dmem.hex>; exit"
```

The full set of implemented functions are:

Name	Arguments	Purpose
raisin64_halt	none	Halts the CPU (**Required** before dumping memory)
raisin64_resume	none	Un-Halts the CPU
raisin64_reset	none	Resets the CPU
raisin64_program	<imem.hex> <dmem.hex>	Programs Instruction and Data memory, resetting CPU
raisin64_dump_dmem	<addr> <size>	Dumps the contents of Data memory
raisin64_dump_imem	<addr> <size>	Dumps the contents of Instruction memory

3.2.1 Getting OpenOCD

As the present time, any modern version of OpenOCD can be used along with the script file for the Raisin64. Official releases are at: http://openocd.org/getting-openocd/ The future Raisin64 version will be located: https://github.com/ChrisPVille/raisin64-openocd

CHAPTER

FOUR

NEXYS 4 DDR REFERENCE IMPLEMENTATION

The Nexys 4 DDR (Using a Xilinx Artix-7 series XC7A100T-1CSG324C) was chosen as the reference implementation due to its copious hardware resources, interactive IO, and sufficient memory for a general purpose operating system to run using the onboard resources. The Raisin64 was connected to memory-mapped peripherals providing access to the LEDs, Switches, and a custom written character oriented VGA controller.

4.1 SoC Peripherals

Included are several trivial IO devices such as the switch and LED interface. These simply wait to be enabled based on the present address and a simple memory map decoder, carrying out the input or output as dictated by the processor output enable and write signals.

IO Memory Map:

Name	Base Address
LED Output	0xFFFFC000_00004000
Switch Input	0xFFFFC000_00008000
VGA Character/Attribute RAM	0xFFFFC000_00040000

Simple IO Control:

```
//////////  IO  //////////
wire led_en, sw_en, vga_en;
memory_map memory_map_external(
    .addr(mem_addr_valid ? mem_addr : 64'h0),
    .led(led_en),
    .sw(sw_en),
    .vga(vga_en)
    );

//As noted in raisin64.v because our IO architecture will need to be completely
//re-written with the introduction of caches, we only support 64-bit aligned
//access to IO space for now.
reg[15:0] led_reg;
always @(posedge clk_dig or negedge rst_n) begin
    if(~rst_n) led_reg <= 16'h0;
    else if(led_en & mem_from_cpu_write) led_reg <= mem_from_cpu;
end

assign LED = led_reg;
```

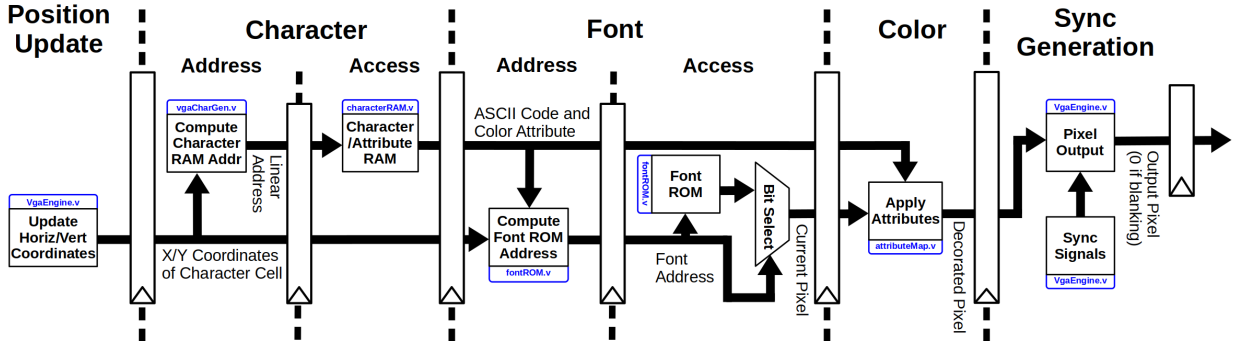
(continues on next page)

(continued from previous page)

```
//SW uses a small synchronizer
reg[15:0] sw_pre0, sw_pre1;
always @(posedge clk_dig or negedge rst_n) begin
    if(~rst_n) begin
        sw_pre0 <= 16'h0;
        sw_pre1 <= 16'h0;
    end else begin
        sw_pre0 <= sw_pre1;
        sw_pre1 <= SW;
    end
end

//Data selection
assign mem_to_cpu_ready = mem_addr_valid;
assign mem_to_cpu = sw_en ? sw_pre0 :
                    vga_en ? vga_dout :
                    64'h0;
```

The VGA controller is a much more complicated device, although it presents a simple interface to the CPU. Each "cell" in the video memory is a combined 16-bit character/attribute word, with the least significant 8-bits containing the ASCII character to draw and the most significant 8-bits containing the character's color attributes.

VGA Controller Block Diagram:



More information is available at https://github.com/ChrisPVille/VGA-CharGen

4.2 Required Hardware

- Nexys 4 DDR (Also known as Nexys A7)
- Bus Blaster (or another OpenOCD compatible JTAG Probe)
- VGA Monitor/Adapter

4.3 Synthesizing the Core

The Vivado 2018.2 project can either be cloned from the project repository (**don't forget** to use the recursive flag), or a pre-packaged release can be downloaded from the release page .

When opening the .xpr in Vivado, it should re-scan the source directories and update its module hierarchy. The project is configured for default non-aggressive implementation options to speed synthesis and place/route. With these

defaults, it should only take one or two minutes to get through implementation on a reasonably fast machine. The resulting utilization should be similar or less than:

Site Type	Used	Fixed	Available	Util%
Slice LUTs	4109	0	63400	6.48
• LUT as Logic	3910	0	63400	6.17
• LUT as Memory	199	0	19000	1.05
• LUT as Distributed RAM	176	0		
• LUT as Shift Register	23	0		
Slice Registers	2363	0	126800	1.86
• Register as Flip Flop	2363	0	126800	1.86
• Register as Latch	0	0	126800	0.00
F7 Muxes	73	0	31700	0.23
F8 Muxes	0	0	15850	0.00
Block RAM Tile	13	0	135	9.63
• RAMB36E1	13	0	135	9.63
• RAMB18	0	0	270	0.00
DSPs	16	0	240	6.67
• DSP48E1 only	16			

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

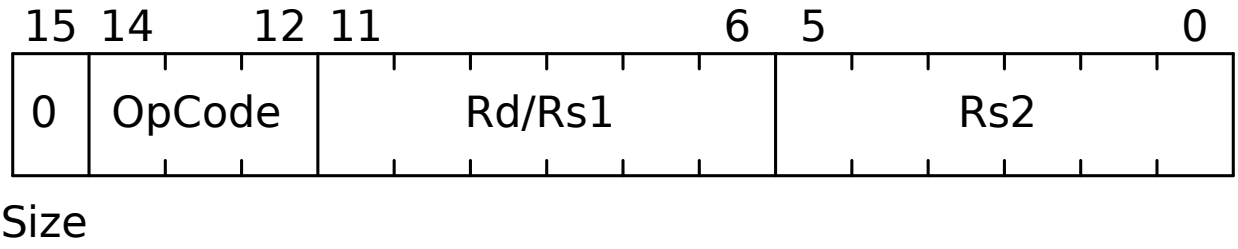
There are 6 instruction formats in Raisin64, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

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.

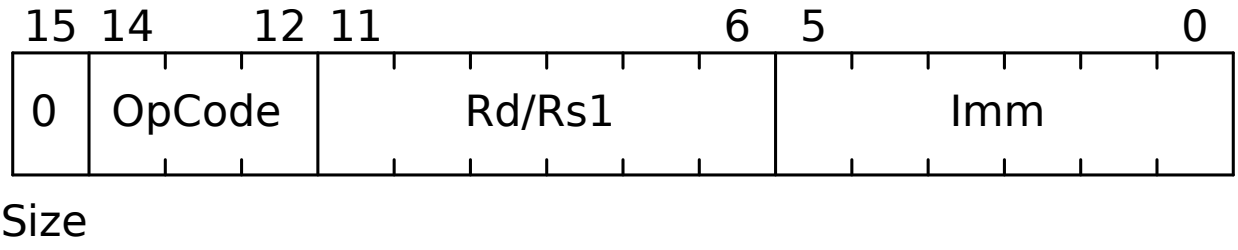
5.1.3 16-bit formats

5.1.3.1 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).

5.1.3.2 16I - 16-bit Immediate Format



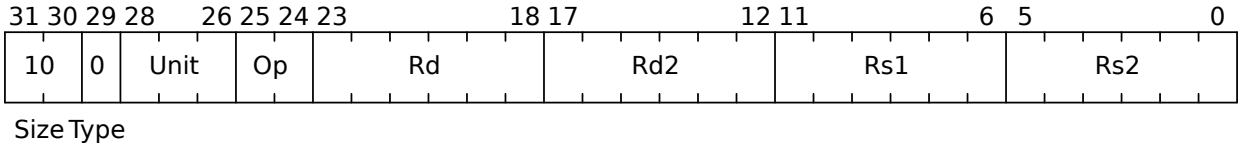
The 16-bit immediate format is used only for ADDI and SUBI, allowing for small increment and decrement operations in a compact format.

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	

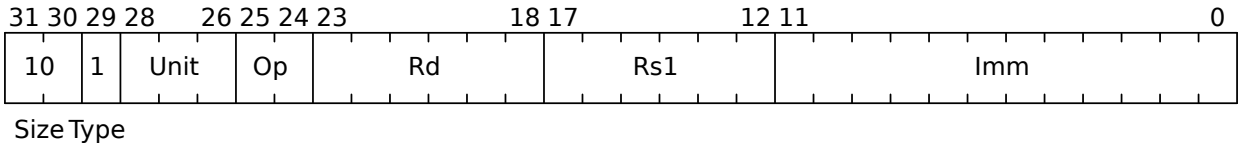
5.1.4 32/64-bit Formats

5.1.4.1 32R - 32-bit Register Format



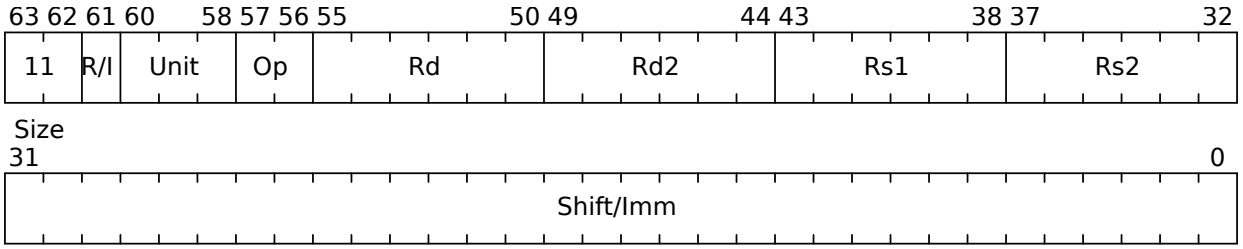
All register type instructions in the Raisin64 are available in 32R format. The only exception of this is the F* FPU call, which also uses the immediate field of the 64S format.

5.1.4.2 32I - 32-bit Immediate Format



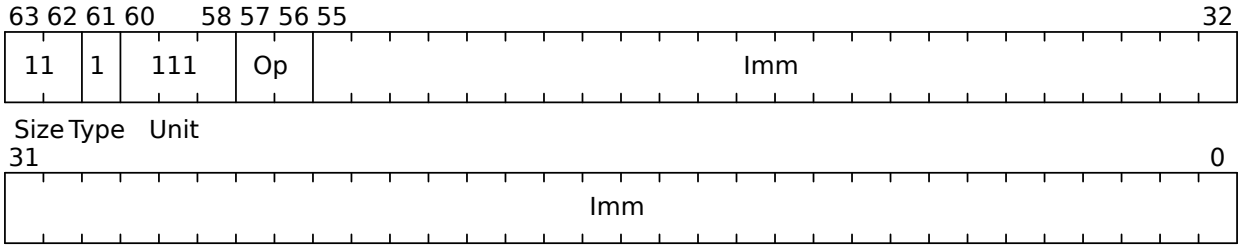
With the exception of JI, JALI, and LUI, all immediate type instructions in the Raisin64 are available in the 32I format with a 12-bit immediate value.

5.1.4.3 64S - 64-bit Standard Format



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)[T2]
		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
		1 - SRAI
		2 - SRLI
	3 - Bitwise Op	0 - ANDI
		1 - NORI

Continued on next page

Table 1 – continued from previous page

R/I	Unit	Op
		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 only

[2] The F* instruction uses the immediate field of 64S to request a specific enumerated service from the FPU. These instructions (and FPU) do not yet exist.

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

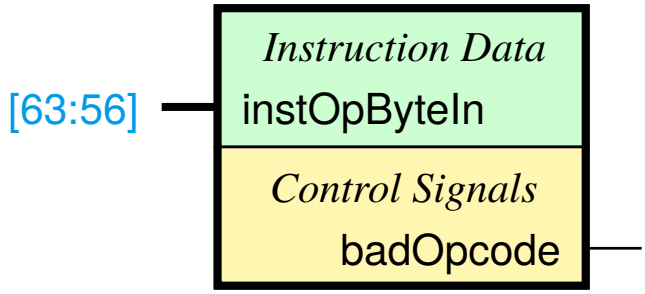


Fig. 1: de_badDetect.v

```
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

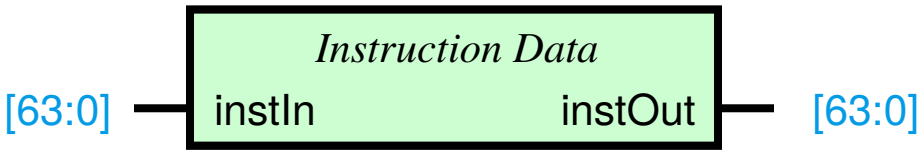


Fig. 2: de_canonicalize.v

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

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

    `include "de_isa_def.vh"

    reg[63:0] instOut_pre;
    assign instOut = instOut_pre;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
        end

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

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

                //Sign-Extended type
                if(instIn[60] || //Most sign-extended Ops (remember, JALI and JI are invalid for 32I)
                    instIn[60:58]==3'h0 || //ADDI/SUBI signed versions sign extend
                    instIn[61:56]==`OP_SLTI || //SLTI and SGTI are signed and sign extend
                    instIn[61:56]==`OP_SGTI)
                       instOut_pre[31:0] = {{20{instIn[43]}},instIn[43:32]};
                //Non Sign-extended type
                else instOut_pre[11:0] = instIn[43:32];

            //32R-Type instructions
            end else begin
                //Set the other operands
                instOut_pre[49:32] = instIn[49:32];

                //No immediate
            end
        end
        //64-bit instructions
        else instOut_pre = instIn;
    end
endmodule
```

5.2.3 de_isa_def.vh

5.2.4 ex_advint.v

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

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

    //# {{data|ALU Data}}
    input[63:0] in1,
    input[63:0] in2,
    output reg[63:0] out,
    output reg[63:0] out2,

    //# {{control|Dispatch Control Signals}}
```

(continues on next page)

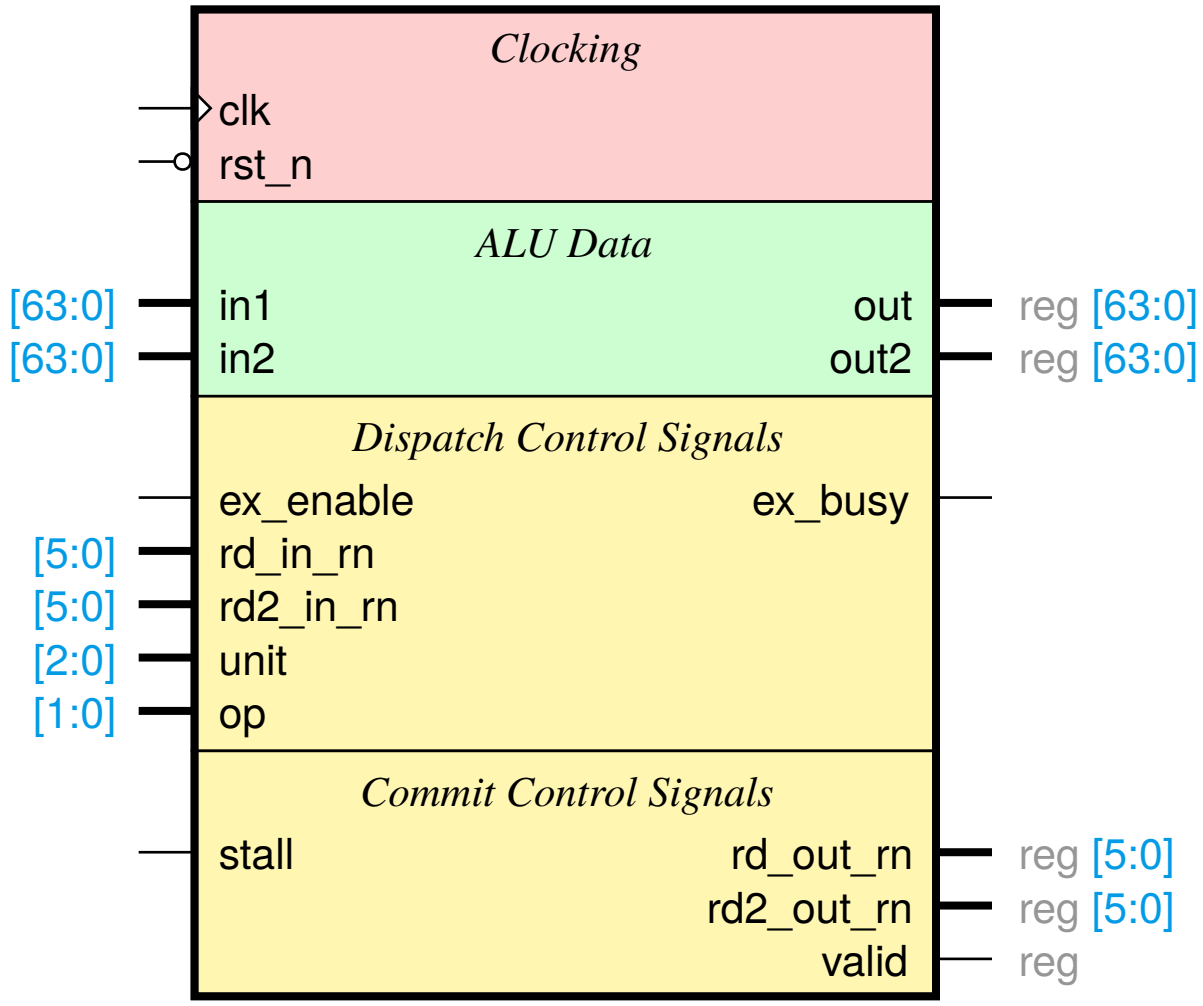


Fig. 3: ex_advint.v

(continued from previous page)

```
    input ex_enable,
    output ex_busy,
    input[5:0] rd_in_rn,
    input[5:0] rd2_in_rn,
    input[2:0] unit,
    input[1:0] op,

    //# {{control|Commit Control Signals}}
    output reg[5:0] rd_out_rn,
    output reg[5:0] rd2_out_rn,
    output reg valid,
    input stall
    );

    wire[63:0] out_pre;
    wire[63:0] out2_pre;

    //We allow the next result to register when we aren't explicitly beign
    //stalled by the next stage (i.e. our result has somewhere to go).
    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            valid <= 0;
            out <= 64'h0;
            out2 <= 64'h0;
            rd_out_rn <= 6'h0;
            rd2_out_rn <= 6'h0;
        end else begin
            valid <= ex_enable;
            out <= out_pre;
            out2 <= out2_pre;
            rd_out_rn <= rd_in_rn;
            rd2_out_rn <= rd2_in_rn;
        end
    end

    initial begin
        if(stall&ex_enable) $error("Told to execute AdvInt when commit was stalled");
    end

    //As this is a one-cycle stage for now, busy is simple
    assign ex_busy = stall;

    ex_advint_s1 ex_advint_s1_1(
        .in1(in1), .in2(in2), .out(out_pre), .out2(out2_pre),
        .enable(ex_enable), .unit(unit), .op(op)
        );

endmodule
```

5.2.5 ex_alu.v

```
//Raisin64 Execute Unit - Integer ALU

module ex_alu(
```

(continues on next page)

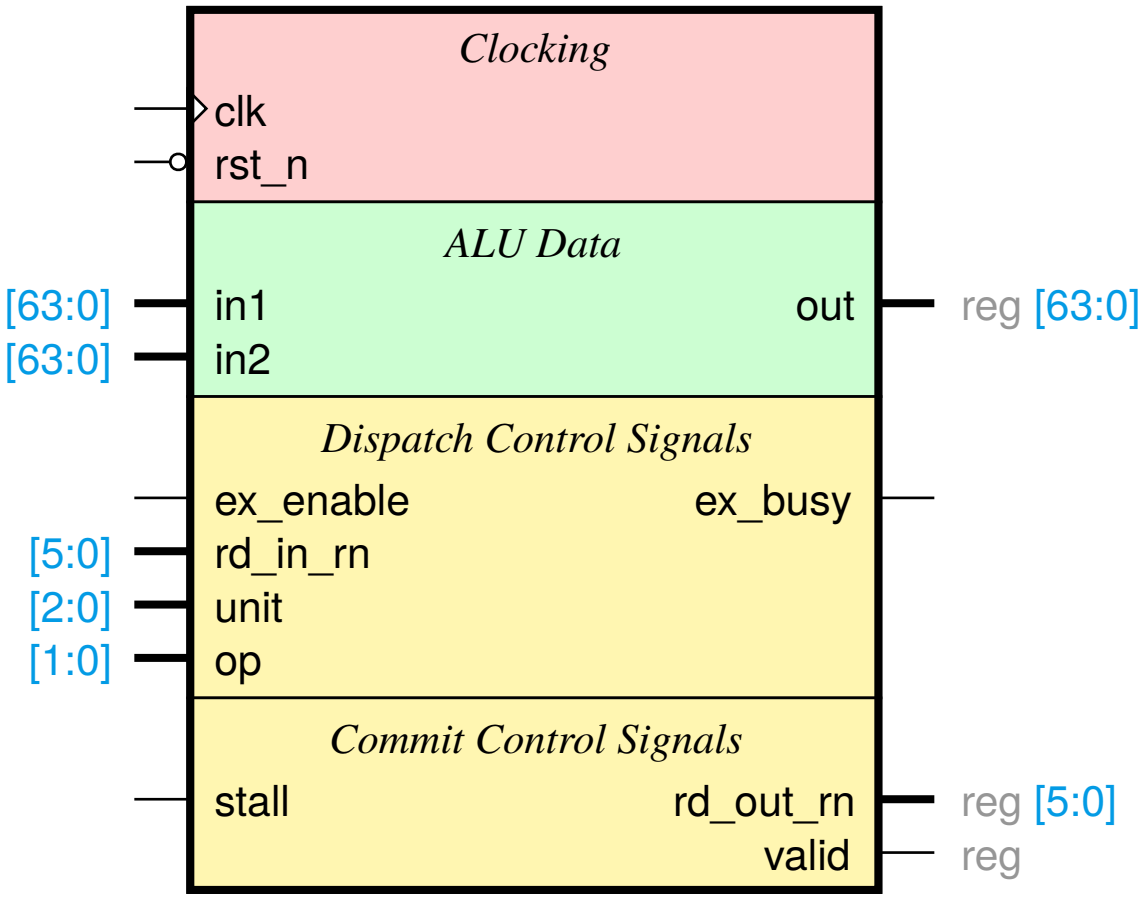

Clocking
clk
rst_n
ALU Data
[63:0]
in1
out
reg [63:0]
[63:0]
in2
Dispatch Control Signals
ex_enable
ex_busy
[5:0]
rd_in_rn
[2:0]
unit
[1:0]
op
Commit Control Signals
stall
rd_out_rn
reg [5:0]
valid
reg

Fig. 4: ex_alu.v

(continued from previous page)

```
    //# {{clocks|Clocking}}
    input clk,
    input rst_n,

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

    //# {{control|Dispatch Control Signals}}
    input ex_enable,
    output ex_busy,
    input[5:0] rd_in_rn,
    input[2:0] unit,
    input[1:0] op,

    //# {{control|Commit Control Signals}}
    output reg[5:0] rd_out_rn,
    output reg valid,
    input stall
    );

    wire[63:0] out_pre;

    //We allow the next result to register when we aren't explicitly beign
    //stalled by the next stage (i.e. our result has somewhere to go).
    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            valid <= 0;
            out <= 64'h0;
            rd_out_rn <= 6'h0;
        end else if(~stall) begin
            valid <= ex_enable;
            out <= out_pre;
            rd_out_rn <= rd_in_rn;
        end
    end

    initial begin
        if(stall&ex_enable) $error("Told to execute ALU when commit was stalled");
    end

    //As this is a one-cycle stage, busy is simple
    assign ex_busy = stall;

    ex_alu_s1 ex_alu_s1_1(
        .in1(in1), .in2(in2), .out(out_pre), .enable(ex_enable),
        .unit(unit), .op(op)
        );

endmodule
```

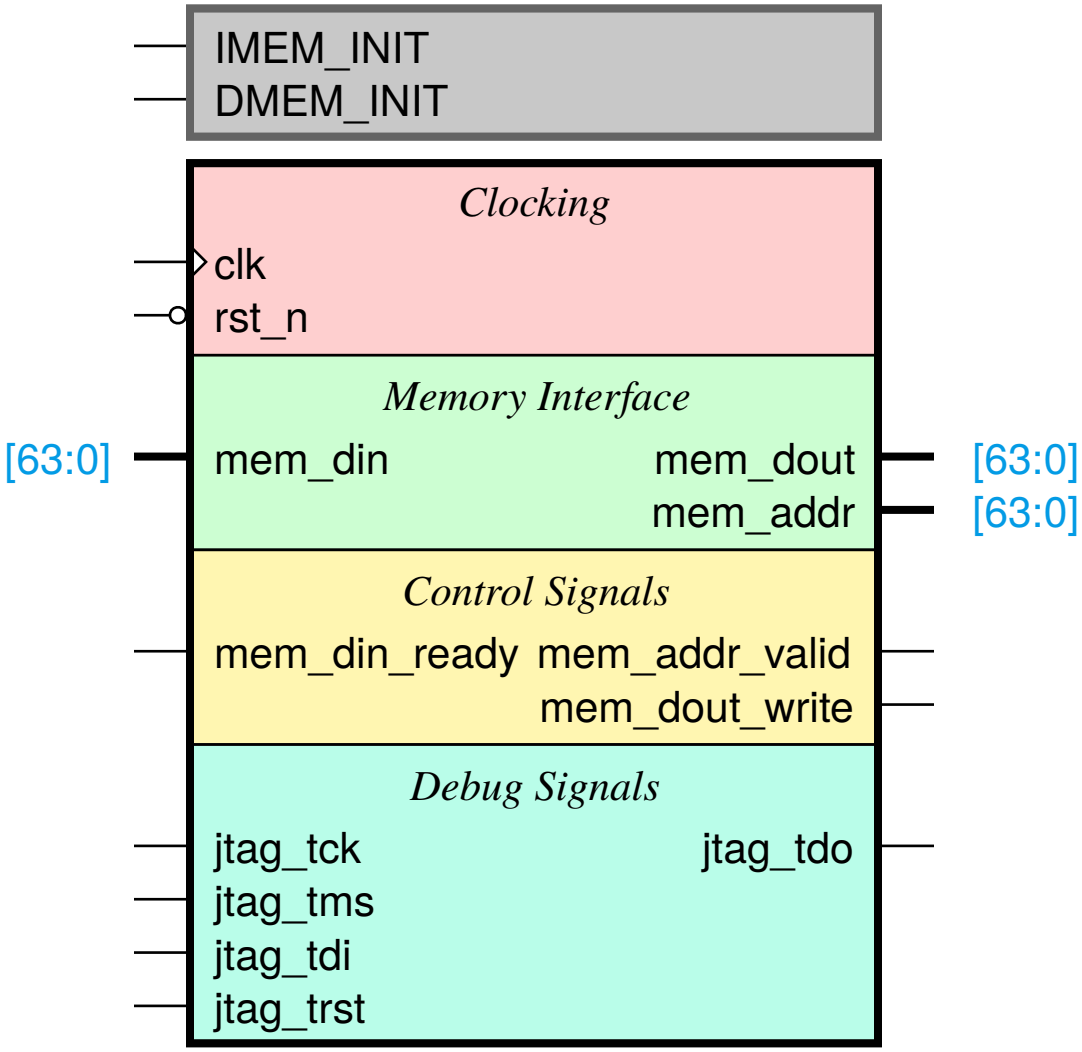


Fig. 5: Raisin64.v

5.2.6 Raisin64.v

```
/*
 * Raisin64 CPU
 */

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

    //# {{data|Memory Interface}}
    input[63:0] mem_din,
    output[63:0] mem_dout,
    output[63:0] mem_addr,

    //# {{control|Control Signals}}
    output mem_addr_valid,
    output mem_dout_write,
    input mem_din_ready,

    //# {{debug|Debug Signals}}
    input jtag_tck,
    input jtag_tms,
    input jtag_tdi,
    input jtag_trst,
    output jtag_tdo
    );

    parameter IMEM_INIT = "";
    parameter DMEM_INIT = "";

    //// Debug Signals ////
    wire dbg_resetn_cpu, dbg_halt_cpu;
    wire cpu_rst_n;
    assign cpu_rst_n = rst_n & dbg_resetn_cpu;

    wire[63:0] dbg_imem_addr;
    wire[63:0] dbg_imem_to_ram;
    wire dbg_imem_ce;
    wire dbg_imem_we;

    wire[63:0] dbg_dmem_addr;
    wire[63:0] dbg_dmem_to_ram;
    wire dbg_dmem_ce;
    wire dbg_dmem_we;


    //////////  Instruction RAM  //////////
    wire[63:0] effective_imem_addr;
    wire[63:0] effective_imem_data_to_cpu;

    reg imem_data_ready;
    wire imem_addr_valid;
    wire[63:0] imem_addr;
    wire[63:0] imem_data;

```

(continues on next page)

(continued from previous page)

```
    assign effective_imem_addr        = dbg_halt_cpu ? dbg_imem_addr : imem_addr;
    assign effective_imem_data_to_cpu = dbg_halt_cpu ? 64'h0 : imem_data;

    always @(posedge clk or negedge rst_n) begin
        if(~rst_n) imem_data_ready <= 0;
        else imem_data_ready <= imem_addr_valid;
    end

    ram #(
        .NUM_BYTES(2*1024),
        .INIT_FILE(IMEM_INIT)
        ) imem (
        .clk(clk),
        .we(dbg_imem_we), .cs(1'b1),
        .write_width(2'h0),
        .addr(effective_imem_addr),
        .data_in(dbg_imem_to_ram),
        .data_out(imem_data)
        );


    //////////  Data RAM  //////////
    wire[63:0] effective_dmem_addr;
    wire[63:0] effective_dmem_to_ram;

    wire[63:0] dmem_addr;
    wire[63:0] dmem_to_ram;
    wire[63:0] dmem_to_cpu;
    wire[63:0] dmem_from_ram;
    wire[1:0] dmem_write_width;
    reg dmem_cycle_complete;
    wire dmem_rstrobe;
    wire dmem_wstrobe;

    wire io_space;

    assign effective_dmem_addr        = dbg_halt_cpu ? dbg_dmem_addr : dmem_addr;
    assign effective_dmem_to_ram      = dbg_halt_cpu ? dbg_dmem_to_ram : dmem_to_ram;

    //TODO For now, the external memory bus is just for data memory.  When the time
    //comes for caches, this will change to the unified external memory bus.
    assign dmem_to_cpu                = io_space ? mem_din : dmem_from_ram;
    assign mem_dout                   = effective_dmem_to_ram;
    assign mem_addr                   = effective_dmem_addr;
    assign mem_addr_valid             = 1;
    assign mem_dout_write             = dbg_halt_cpu ? dbg_dmem_we : dmem_wstrobe;

    //Because the memory interface will change dramatically in the next revision, there
    //is no reason to create special logic to handle misaligned accesses into data space
    //in case an IO unit requires it (the ram modules handle this condition internally).
    //Instead we simply state misaligned IO access it is unsupported (for now).
    always @(*) begin
        if((dmem_rstrobe|dmem_wstrobe) & io_space & |dmem_write_width & ~clk) begin
            $display("Unaligned data IO access not supported in this revision");
```

(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_
↪wstrobe|dmem_rstrobe|dbg_dmem_ce)),
134         .write_width(dmem_write_width),
135         .addr(effective_dmem_addr),
136         .data_in(effective_dmem_to_ram),
137         .data_out(dmem_from_ram)
138         );
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
158     //////////  JTAG Module  //////////
159     debug_control debug_if(
160         .jtag_tck(jtag_tck),
161         .jtag_tms(jtag_tms),
162         .jtag_tdo(jtag_tdo),
163         .jtag_tdi(jtag_tdi),
164         .jtag_trst(jtag_trst),
165         .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_resetn_cpu(dbg_resetn_cpu),
180         .cpu_halt_cpu(dbg_halt_cpu)
181         );
182
183 endmodule
```

5.2.7 rf_reg.v

CHAPTER

SIX

FOOTNOTES