Raisin64 Documentation

Release 0.2a

Christopher Parish

Dec 13, 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.

Major features of the Raisin64 include:

Bits 64-bit

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

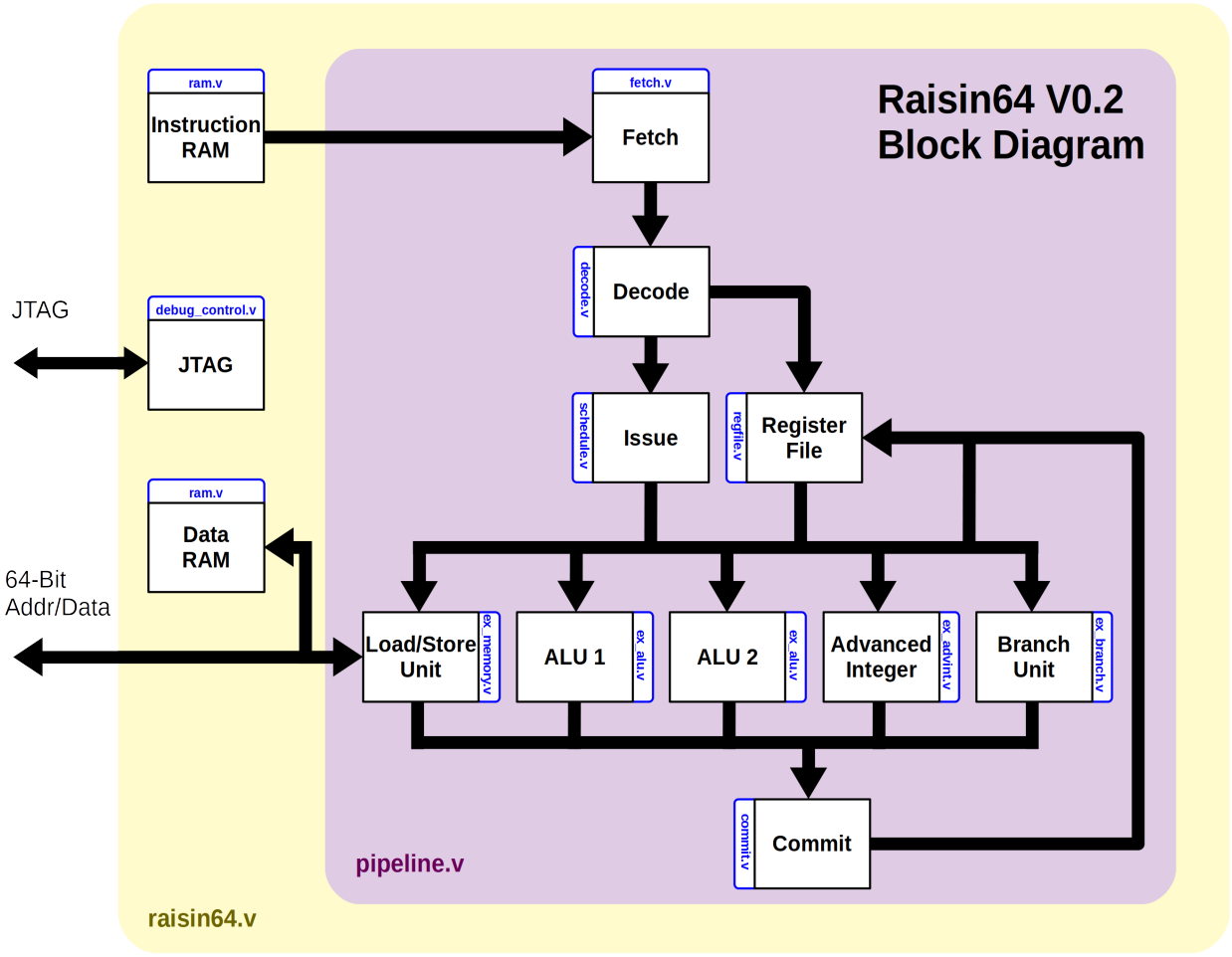
Design RISC (Harvard Architecture[2])

Type Register-Register

Branching Compare and Branch

Endianness Big

Registers 63 (R0 = 0)



[1] Raisin64 will execute 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.

The various repositories hosting the source-code described in this document are available here:

Metaproject	https://github.com/ChrisPVille/raisin64
CPU RTL	https://github.com/ChrisPVille/raisin64-cpu
Binutils	https://github.com/ChrisPVille/raisin64-binutils
Nexys4DDR Example	https://github.com/ChrisPVille/raisin64-nexys4ddr
Binutils	https://github.com/ChrisPVille/raisin64-binutils
OpenOCD	https://github.com/ChrisPVille/raisin64-openocd
Docs Source	https://github.com/ChrisPVille/raisin64-docs
PDF Documentation	https://raisin64.fsys.io/_build/latex/Raisin64.pdf

CHAPTER

ONE

RAISIN64 CPU

1.1 Purpose

The Raisin64, like most computer processors, is a collection of various processing elements and memories, creating what is presently a pipelined 64-bit Harvard RISC architecture. While the initial semester of work focused on getting the design off the ground: designing the instruction set, laying out the execution pipeline, preparing the tools, etc., the eventual goal is to create a CPU capable of running a modern general purpose operating system. This also includes porting all the required debug utilities, assembler, compiler, and other tools necessary to accomplish that goal.

Given the relative paucity of new ISAs, there seems to be little in the way of academic coursework and technical documentation on the bootstrapping process which must occur to make a new architecture useful. This project is the result of those observations, and provides me with experience beyond just computer architecture, including those software and hardware tasks necessary to actually do something useful with a CPU.

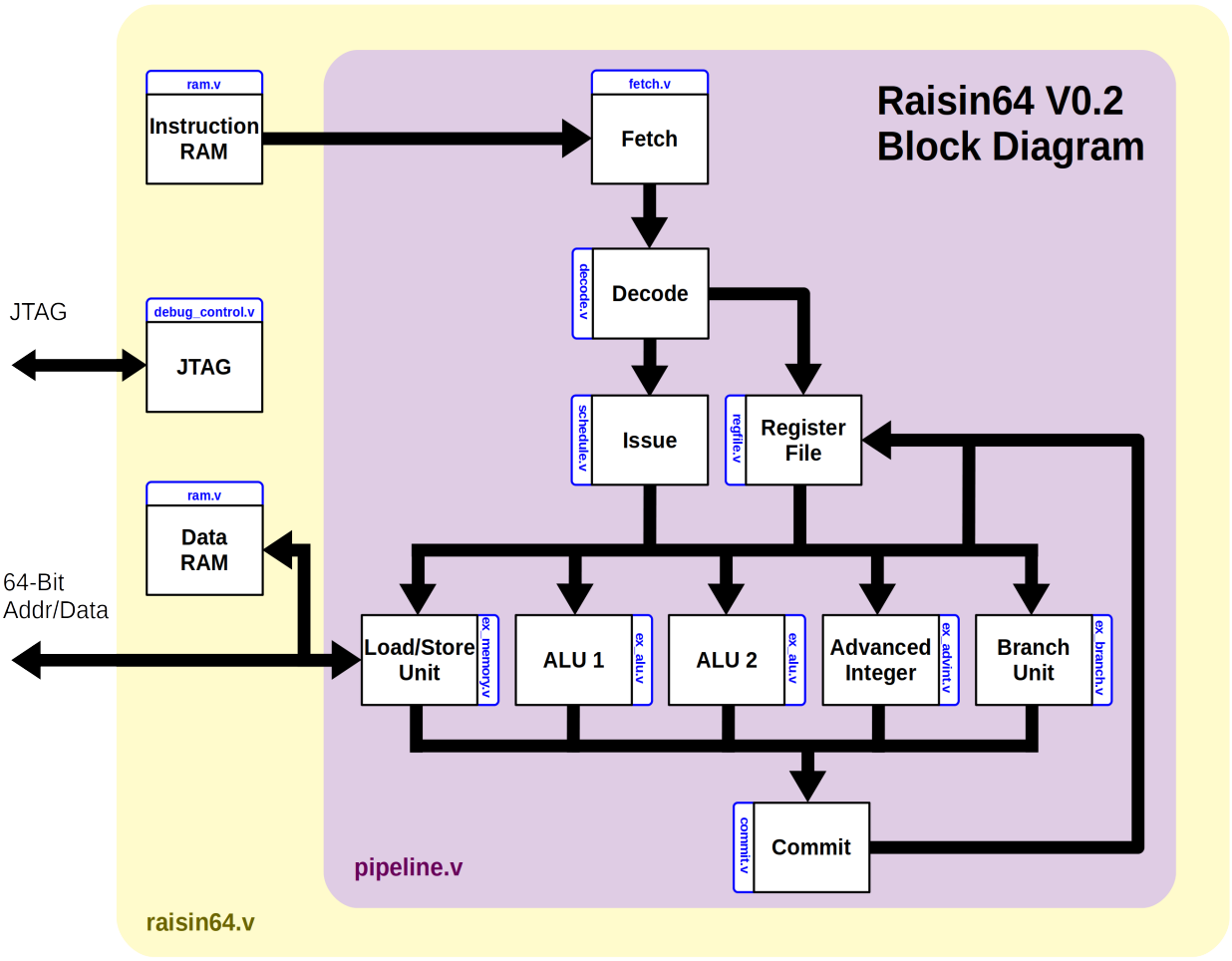
All of that said, the Raisin64 is an initial attempt at understanding pipelined processors, the trades between register count, opcode density, and speed, as well as being a platform for further experimentation. Envisioned as a pure 64-bit machine, the Raisin64 has no legacy instruction set to support and can start as a clean slate.

Anticipating that an out-of-order design with register renaming would be too much work for a semester long project, the Raisin64 ISA has a large architectural register set with 63 64-bit registers instead (Register 0 is always 0). These additional registers have consequences for the instruction format, requiring 6 bits to represent. While a 64-bit instruction word can easily store that, a fixed 64-bit instruction size would be immensely wasteful in terms of memory and cache utilization.

I decided to create a compact instruction format that allows certain instructions to have 16-bit representations and nearly all to have 32-bit representations, with only a few actually requiring the full 64-bit word (such as *Jump Immediate*). Of course, a fixed instruction word is convenient, so the Raisin64's decode stage is designed to expand the 16 and 32-bit instructions into the full 64-bit canonical form as it arrives. This also allows savings on cache and memory while having a simpler internal processing pipeline.

Full details on the instruction word formatting are documented in the *Raisin64 Instruction Set* section.

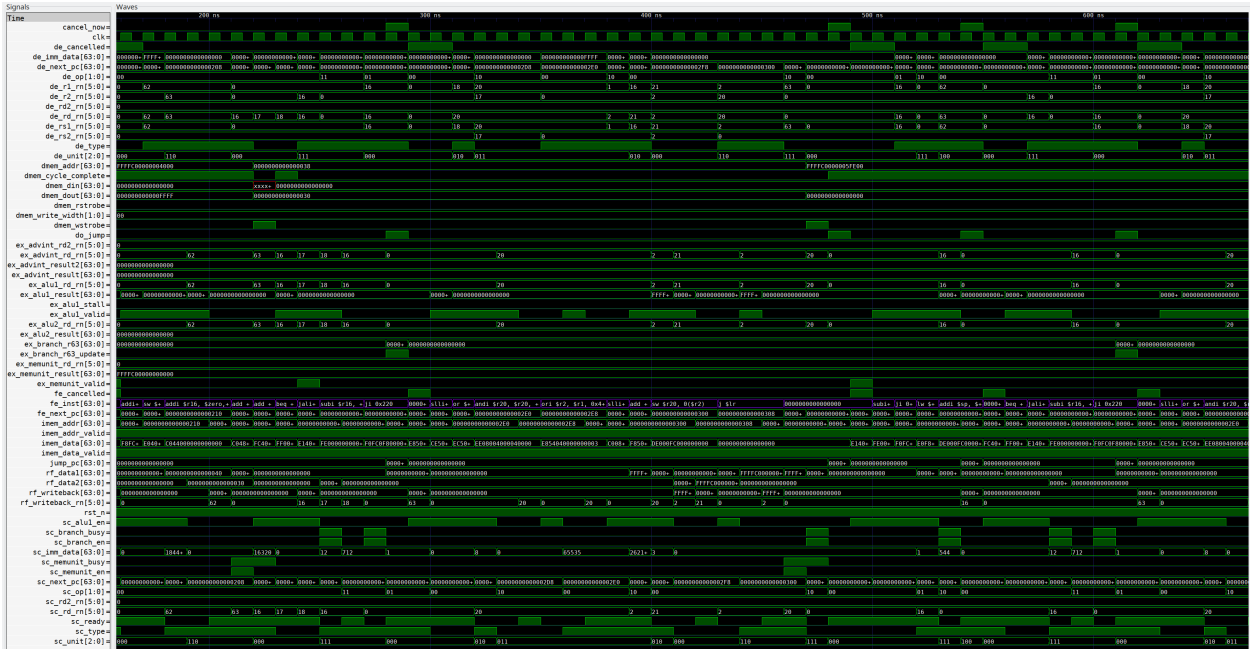
1.2 Pipeline Stages



The Raisin64 pipeline connects together the processing elements detailed in later sections. Having seen several academic and open-source processor designs, the processing pipeline tends to be one of the more confusing modules to look at, not because of algorithmic complexity but due to the large number of signals moving around the design.

Special effort was made to minimize the number of delay registers and extraneous signals between the various pipeline stages, keeping the design understandable and relatively easy to debug.

Below is an example of the Raisin64's pipeline executing the *VGA Demo Program*:



To assist with organization, signals in *pipeline.v* are generally prefixed with the module that generated the signal:

Stage:	Prefix:
Fetch	fe_*
Decode	de_*
Register File	rf_*
Scheduler	sc_*
Execution Unit	ex_unitName_*

1.2.1 Fetch Unit

The *fetch unit* is responsible for maintaining the program counter, calculating the next linear instruction's address based on the size of the present instruction, and muxing in the jump/branch destination address.

Fetch is relatively straightforward although it does need to support stalls from both the memory interface (as the data may not be ready) and from deeper in the pipeline (should an instruction need to wait for execution resources).

1.2.2 Decode Unit

The *decode unit* performs a hardwired conversion between the input instruction word and the control signals used by later pipeline stages. Starting with one or more instruction words left-aligned from the fetch unit, the decode unit passes the instruction simultaneously to the *instruction canonicalizer* for conversion into the native 64-bit instruction word and to an *invalid opcode detector* which is unused (owing to the lack of interrupts/exceptions).

Presently, the decode unit decodes a single instruction at a time, limiting the issue rate of the processor to 1 at most. Given the overall complexity of the design for a single semester project, it was decided to defer the multi-issue capability until next semester when the caches and fetch system will be written and adapted to wider access.

The decode unit also selects the two source register numbers for any instruction. While these generally map exactly to the `$rs1` and `$rs2` fields, opcodes like *Branch if Equal* use two registers and the immediate field. To avoid creating another instruction format, the usual *32R* format can be used, directing the register file to load from `$rd` instead.

There is no scenario where more than two registers are loaded, so the decode unit publishes zero, one, or two register numbers for the register file to load in the next stage.

1.2.3 Register File

The *register file* is a 63 entry, 64-bit dual read-port, single write-port RAM. The processing pipeline was architected to allow for a single cycle of read latency from the register file. This allows the otherwise large register file to map to higher density memories like distributed RAM or block RAM in FPGA implementations.

The register file is also designed to allow write values to fall through to the read port should another instruction be requesting the same register value. This data forwarding can save an otherwise wasted cycle.

1.2.4 Schedule Unit

The *scheduler*, or issue unit as it is conventionally known, runs concurrent to the register file on a given time-step. This allows the register file time to gather the data required for an operation while the scheduler is making a decision whether or not to issue that instruction.

As execution units are allowed to take more than one cycle to complete, the scheduler tries to issue instructions up to the point where there are either no free execution resources or one of the source operands is going to be written by an in-progress instruction that hasn't finished.

This is accomplished by marking those destination register numbers busy when they are issued and unmarking them when they are written back to the register file. A limited non-speculating in-order issue with out-of-order execution is a side-effect of the arrangement assuming register numbers don't overlap. Consider the following:

```
lw $r2, ($r1)       #Load data at address $r1 into $r2
add $r3, $r4, $r5   #Add $r4 and $r5, storing in $r3
add $r4, $r5, $r6
```

It is very likely that the ADD instructions will finish executing before the LW. The scheduler will identify LW affects only `$r2`, and that it is not used in the subsequent instructions. Because instructions are always *issued* in order, the re-use of `$r4` is not a problem here either. It will always be read from the register file before the final ADD instruction is issued.

1.2.5 Execution Units

The execution units in the Raisin64 are entirely independent modules with a standard set of control signals:

```
input[63:0] in1,        //Input data A
input[63:0] in2,        //Input data B
output[63:0] out,       //Output data

input ex_enable,        //Execute now
output ex_busy,         //We are busy and cannot accept data
input[5:0] rd_in_rn,    //Destination Register Number
input[2:0] unit,        //Unit field from the instruction
input[1:0] op,          //Op field from the instruction

output[5:0] rd_out_rn, //Register Number to the commit unit
output valid,           //Output data is valid
input stall             //Commit unit is stalling us
```

The Raisin64 does allow execution units to have two output registers (which is used notably by the *Advanced Integer Unit*), but most only have one.

1.2.5.1 Integer Unit

The *integer unit/ALU* is the registering wrapper around the *combinational ALU implementation*. As this project was not focused on computer arithmetic and with the knowledge that the design would generally target FPGAs, it was decided to leave the Verilog operators for addition and subtraction. Synthesizers can be quite good at using dedicated hardware IP or putting down whatever adder implementation will best satisfy the speed and area constraints.

1.2.5.2 Advanced Integer Unit

The *advanced integer unit* is the registering wrapper around the *combinational advanced integer unit implementation*. Again, for similar reasons, the math operations were left as Verilog operators in the hopes that they would map nicely to onboard hard-IP present in the FPGA. The multiplier did so, mapping to a DSP block containing a multiply-accumulate unit. Unfortunately, there is no division hardware present on the FPGA family used for evaluation of the Raisin64.

Future Work

This will need to be converted to a pipelined division unit (or reciprocal/multiplication) at some point in the future, but for now division is disabled as execution pipeline stalls are already proven and the division instruction is ancillary to the project's goals.

1.2.5.3 Branch Unit

The *branch unit* itself is internally simple although the implications for the pipeline are complicated. The branch unit operates in either branch or jump mode, with jump being a trivial distillation of the branching mode. The unit accepts two input words and an immediate displacement value as well as a delayed version of the next linear program counter from the fetch unit. If the two words are equal, the branch unit adds the displacement to the program counter during its execution cycle.

On the next cycle, when results are typically presented to the commit unit, the branch unit will present `$r63` if appropriate for linking, and it will also signal the pipeline via `do_jump` that a jump is being issued. This causes the pipeline to flush currently fetched and decoded instructions to a NOP value, canceling any unissued instructions.

Because it is desirable to allow the branch unit to take an arbitrary length of time calculating whether or not to jump, the scheduler avoids issuing any instructions after a jump until it has completely resolved. The scheduler can be changed to take advantage of the present guaranteed one-cycle branch calculation, but this approach allows for easier experimentation.

1.2.5.4 Memory Unit

The *memory unit* is the pipeline's only window into the data space. Having a separate memory interface, the memory unit handles all load and store operations, calculating the effective address after adding the offsets, presenting the addresses on the bus, waiting for a response, and masking/sign-extending as required by the instruction. It then returns the result to the commit unit.

Future Work

Currently designed with a minimum execution time of 3 cycles, the memory unit could be further optimized to reduce latency given more analysis. The offset calculation and masking/sign-extension were intentionally put in their own stages preemptively for performance reasons which may ultimately be unnecessary.

1.2.6 Commit Unit

As results can arrive from many (possibly all) execution units simultaneously, the *commit unit* serves as a buffer between those results and the register file. This eliminates the need for a multi-ported write into the register file. The exact order of the writes is non-deterministic based on the present state of the commit unit's writeback engine. This does not present data consistency problems given that the scheduler has already taken care of resolving dependencies between the registers.

While the commit unit can only write back one register per cycle, it will continue to do so every cycle until empty, allowing it to drain faster than it will fill.

Future Work

More analysis could be done on whether this is provably the case, but the commit unit will need to be rewritten to support precise interrupts and exceptions, allowing for an orderly (or at least traceable) change of processor state.

1.3 Debug Controller

The JTAGlet is a JTAG TAP written from scratch to allow for easy interfacing between a parallel interface (such as a processor bus) and hardware debug probes. Sitting between Raisin64 and the JTAGlet JTAG TAP is *debug_control.v*. This debug controller exposes JTAG registers from the TAP to the rest of the processor, allowing the debug controller to take over main memory for programming and inspection as well as halt and reset the CPU.

This capability has several advantages. First, it allows for reprogramming the Raisin64 while it's running on an FPGA without waiting for re-synthesis due to a software change. Second, it prevents the synthesizer from optimizing out parts of the processor design that are not reachable with the program preloaded in the instruction RAM.

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 memory mapped at the virtual addresses the process expects.

[1] https://www.kernel.org/doc/Documentation/nommu-mmap.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 Modified 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 will be accessible in a kernel-mode unmapped 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 - 0xFFFFE000_00000000	Kernel-Mode Mapped
0xFFFFDFFF_FFFFFFFF - 0xFFFFC000_00000000	Kernel-Mode Unmapped
0xFFFFBFFF_FFFFFFFF - 0x00004000_00000000	Invalid
0x00003FFF_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:

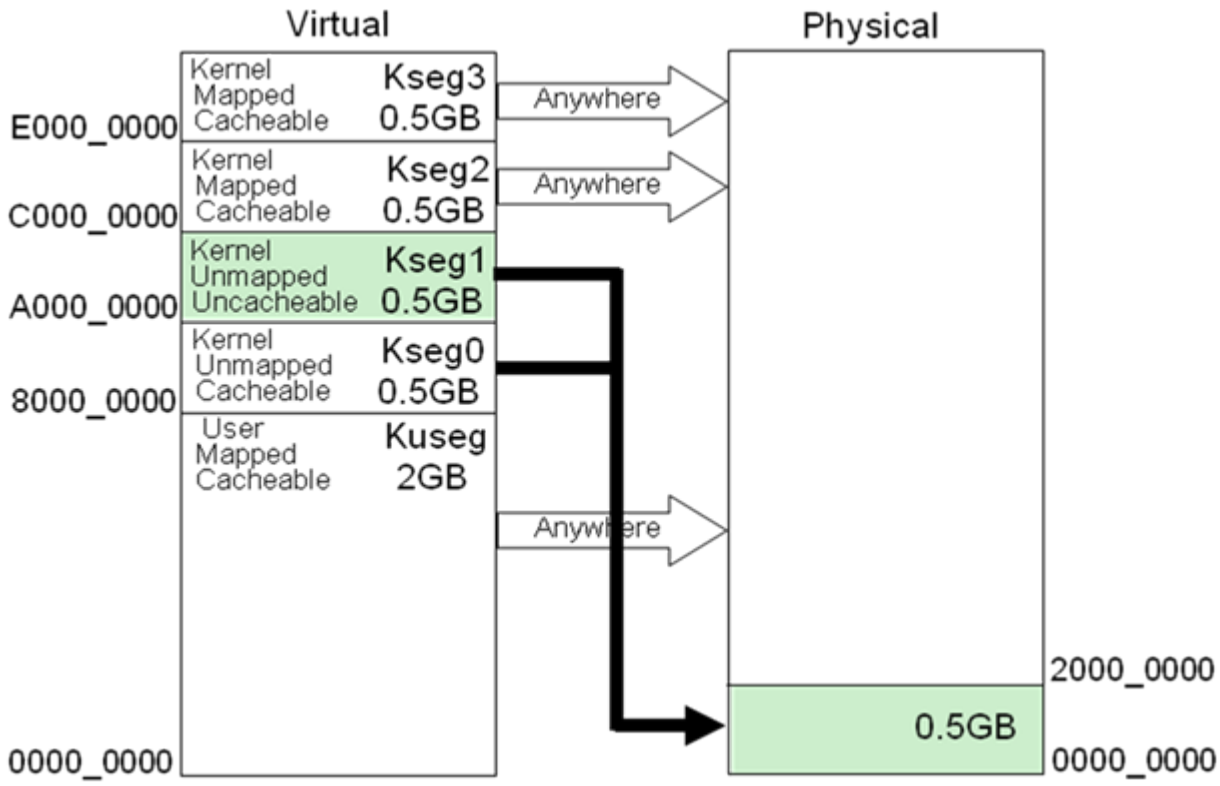


Fig. 1: From ARM AN235 Section 3.4[3]

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 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 will 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 the 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 and flushing their content on a context-switch will only affect 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 but high speed memory available on an FPGA 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

CHAPTER

TWO

CODE SNIPPETS AND SOFTWARE

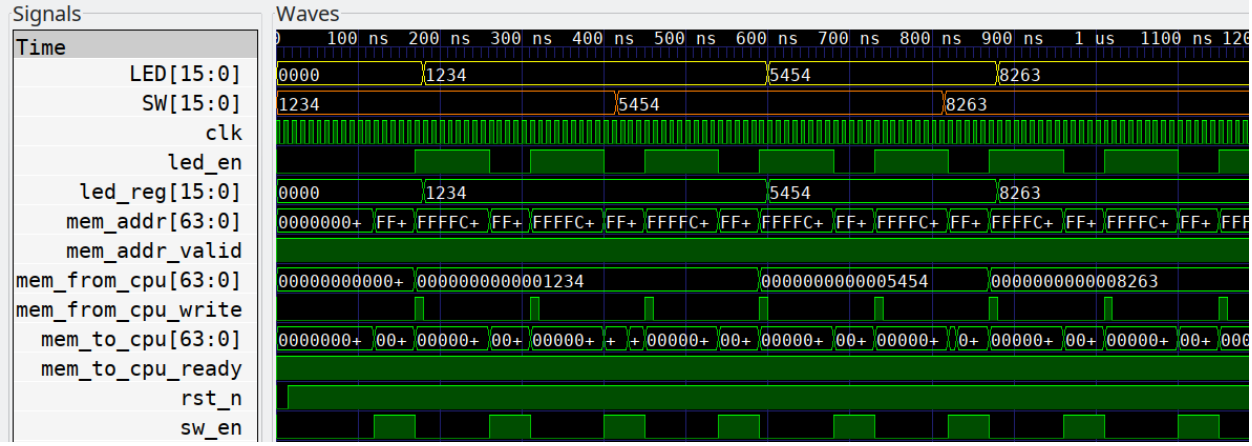
Each instruction detail page contains an example of how to use that specific opcode which will not be repeated here. Instead, a few simple programs will presented that work with the *Nexys 4 DDR Reference Implementation*, demonstrating aspects of the ISA or use of the hardware.

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

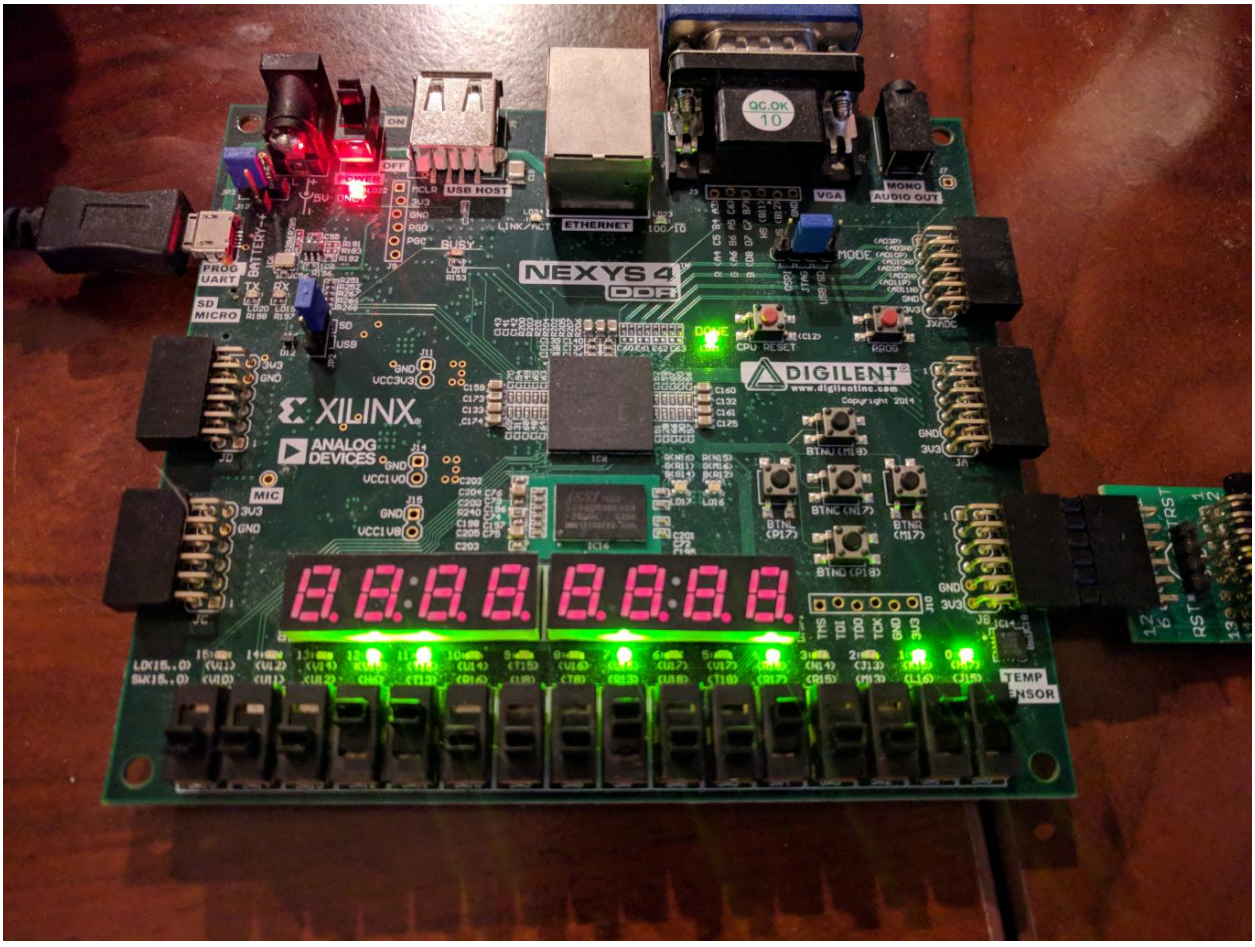
2.1 Switch to LED

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.

```
.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 is 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 on the lower half of the display.

The assembly demonstrates the use of a stack, as well as useful GNU assembler syntax 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
hello_str: .asciz "Hello, World!"
greet_str: .asciz "Greetings from "

```

(continues on next page)

(continued from previous page)

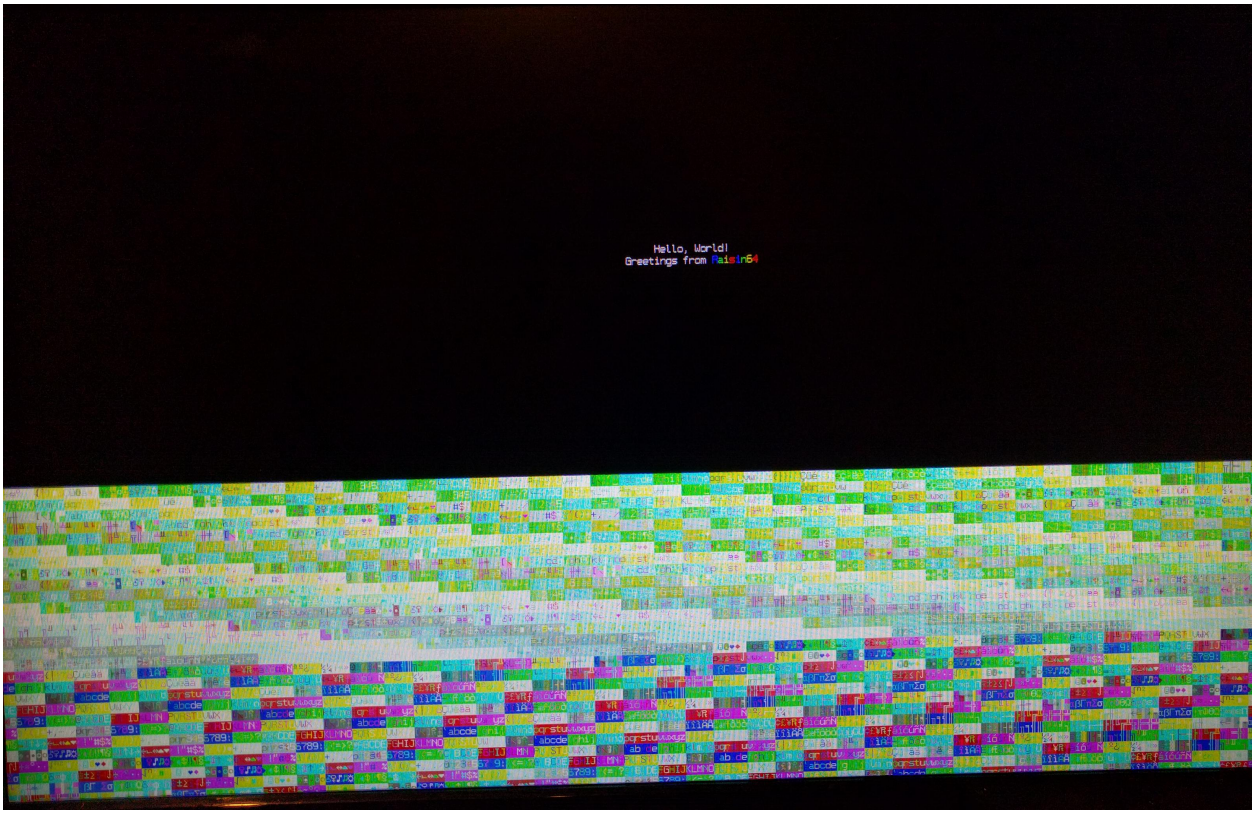
```
.align 9 #Fill 512

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

(continues on next page)

(continued from previous page)

```
    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
    j    $lr

.align 11 #Fill 2K
```

Given the repetitive nature of drawing characters, the simulation is simultaneously un-interesting and overwhelming. Suffice it to say, it leads to a colorful demo.



CHAPTER

THREE

TOOLS

As a completely new computer architecture and instruction set, there 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 given the eventual goal to run a general purpose operating system, GCC will be required. GCC leverages binutils, a collection of assembly tools, object file manipulation utilities, as well as a powerful linker.

Using the preliminary instruction set defined early in the semester, binutils was ported to the Raisin64 ISA while the processor was still being developed. With 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.

The software architecture of the template was noted for it's reasonable size (ISA 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 to 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` style register numbering and a `opcode $dest, $src1, $src2` instruction format.

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>
```

This 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, prints, and extracts the `.text` and `.data` sections (containing the instruction and data memories respectively). Finally, it converts the converting the output files from hex to ASCII using the xxd utility. The result is suitable for the `$readmemh` Verilog command:

```
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 home grown JTAG controller (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 the target 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 accessible at https://github.com/ChrisPVille/raisin64-cpu/blob/master/support/jtag/raisin64_nodeps_openocd.cfg. Although it is currently configured for a Bus Blaster v3, it 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 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.

The example project is accessible at https://github.com/ChrisPVille/raisin64-nexys4ddr

4.1 SoC Peripherals

Included are several trivial IO devices such as the switch and LED interface. These wait to be enabled based on the current address and a simple memory map decoder, carrying out the input or output as dictated by the processor's 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;
```

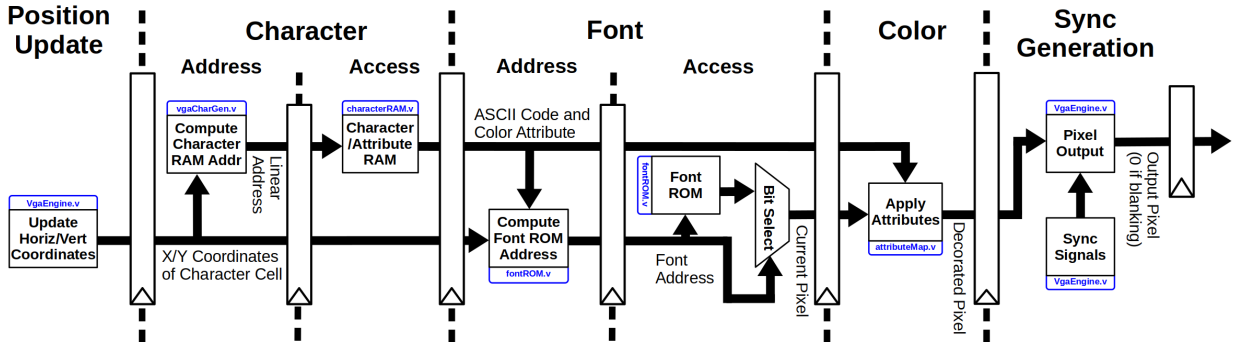
(continues on next page)

(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 *Decode Unit*) 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

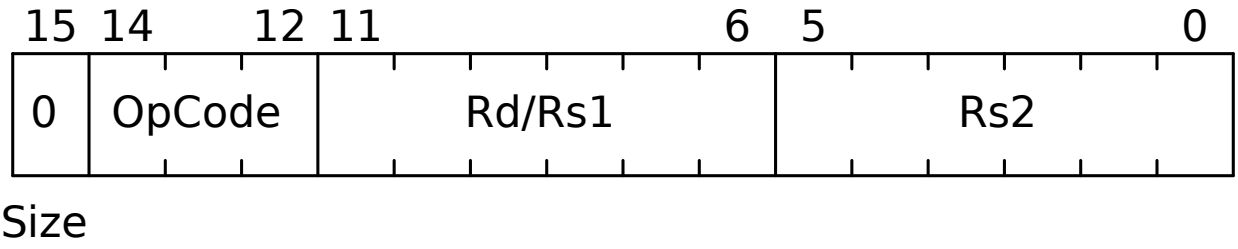
There are 6 instruction formats in Raisin64, register and immediate type 16-bit formats (16R and 16I), register and immediate type 32-bit formats (32R and 32I), and a combined register and immediate 64-bit format (64S) as well as a jump format (64J).

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.

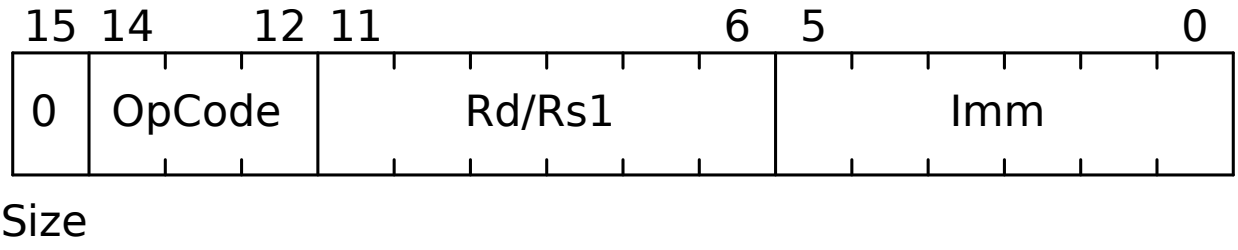
5.1.3 16-bit formats

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



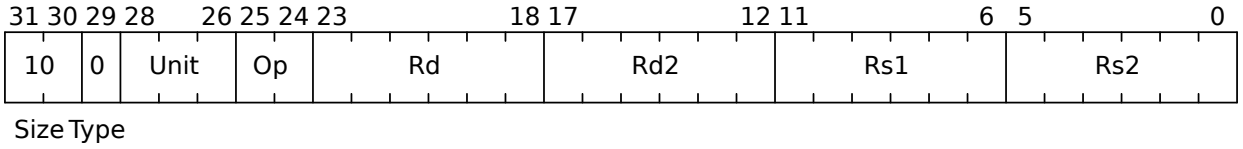
The 16-bit immediate format is used only for ADDI and SUBI, allowing for small increment and decrement operations in a compact format.

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	

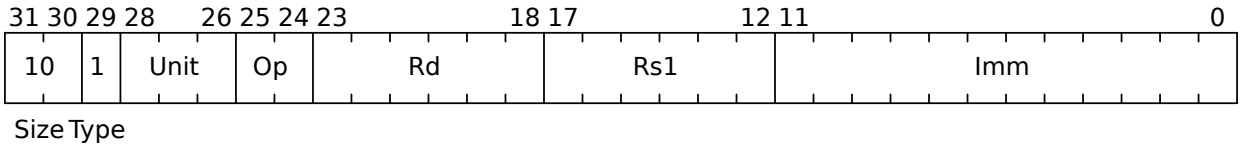
5.1.4 32/64-bit Formats

5.1.4.1 32R - 32-bit Register Format



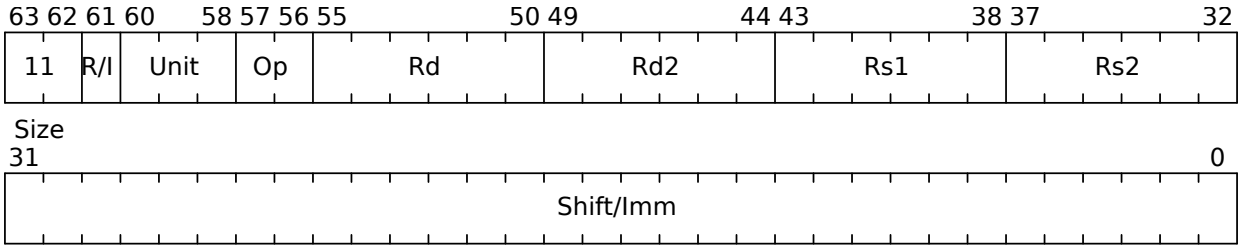
All register type instructions in the Raisin64 are available in 32R format. The only exception of this is the F* FPU call, which also uses the immediate field of the 64S format.

5.1.4.2 32I - 32-bit Immediate Format



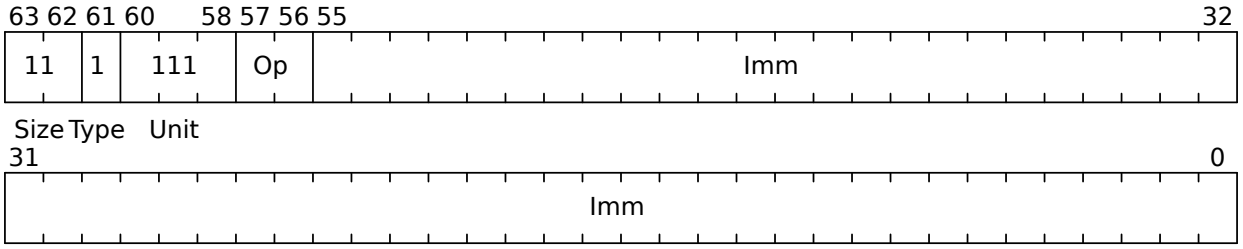
With the exception of JI, JALI, and LUI, all immediate type instructions in the Raisin64 are available in the 32I format with a 12-bit immediate value.

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)[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
		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

ANDs `$rs1` with `$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

ANDs `$rs1` with an immediate value, placing the result in `$rd`.

Usage

```
and $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 `$rs1` register is equal to the `$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 `$zero` with itself.

Usage

```
beq $rd, $rs1, 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 `$rs1` register is equal to the `$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 `$zero` with itself.

Usage

```
beqal $rd, $rs1, 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

```
divu $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

FPU Call (unimplemented)

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 `$rs1`.

Usage

```
j $rs1
```

Operation

```
pc = rs1;
```

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 `$rs1`, placing the return address in `$r63`.

Usage

```
jal $rs1
```

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

Loads 16-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
l16 $rd, imm($rs1)
```

Operation

```
rd[63:16] = 0;
rd[15:8]  = *(rs+imm);
rd[7:0]   = *(rs+imm+1);
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

Loads 16-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
l16s $rd, imm($rs1)
```

Operation

```
rd[63:16] = sign_bit(*(rs+imm));
rd[15:8]  = *(rs+imm);
rd[7:0]   = *(rs+imm+1);
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

Loads 32-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
l32 $rd, imm($rs1)
```

Operation

```
rd[63:32] = 0;
rd[31:24] = *(rs+imm);
rd[23:16] = *(rs+imm+1);
rd[15:8]  = *(rs+imm+2);
rd[7:0]   = *(rs+imm+3);
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

Loads 32-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
l32s $rd, imm($rs1)
```

Operation

```
rd[63:32] = sign_bit(*(rs+imm));
rd[31:24] = *(rs+imm);
rd[23:16] = *(rs+imm+1);
rd[15:8]  = *(rs+imm+2);
rd[7:0]   = *(rs+imm+3);
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

Loads 8-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
l8 $rd, imm($rs1)
```

Operation

```
rd[63:8] = 0;
rd[7:0]  = *(rs+imm);
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

Loads 8-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
l8s $rd, imm($rs1)
```

Operation

```
rd[63:8] = sign_bit(*(rs+imm));
rd[7:0]  = *(rs+imm);
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

Loads the immediate value into the upper 32 bits of `$rd`.

Usage

```
lui $rd, imm
```

Operation

```
rd = imm << 32;
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

Loads 64-bit word from address `$rs1` offset by a signed immediate value and places it into `$rd`.

Usage

```
lw $rd, imm($rs1)
```

Operation

```
rd[63:56] = *(rs+imm);
rd[55:48] = *(rs+imm+1);
rd[47:40] = *(rs+imm+2);
rd[39:32] = *(rs+imm+3);
rd[31:24] = *(rs+imm+4);
rd[23:16] = *(rs+imm+5);
rd[15:8]  = *(rs+imm+6);
rd[7:0]   = *(rs+imm+7);
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

Multiplies registers `$rs1` and `$rs2`, and places the results in `$rd` and `$rd2`, treating operands as 2's complement signed.

Usage

```
mul $rd, $rd2, $rs1, $rs2
```

Operation

```
rd = bottom_64bits(rs1 * rs2);
rd2 = upper_64bits(rs1 * rs2);
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

Multiplies registers `$rs1` and `$rs2`, and places the results in `$rd` and `$rd2`, treating operands as unsigned.

Usage

```
mulu $rd, $rd2, $rs1, $rs2
```

Operation

```
rd = bottom_64bits(rs1 * rs2);
rd2 = upper_64bits(rs1 * rs2);
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

NORs `$rs1` with `$rs2`, placing the result in `$rd`.

Usage

```
nor $rd, $rs1, $rs2
```

Operation

```
rd = ~(rs1|rs2);
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

NORs `$rs1` with an immediate value, placing the result in `$rd`.

Usage

```
nori $rd, $rs1, imm
```

Operation

```
rd = ~(rs1|imm);
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

ORs `$rs1` with `$rs2`, placing the result in `$rd`.

Usage

```
or $rd, $rs1, $rs2
```

Operation

```
rd = rs1 | rs2;
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

ORs `$rs1` with an immediate value, placing the result in `$rd`.

Usage

```
ori $rd, $rs1, imm
```

Operation

```
rd = rs1 | imm;
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

Store least significant 16 bits of `$rd` into the address located in `$rs1` offset by a signed immediate value.

Usage

```
s16 $rd, imm($rs1)
```

Operation

```
*(rs+imm)   = rd[15:8];
*(rs+imm+1) = rd[7:0];
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

Store least significant 32 bits of `$rd` into the address located in `$rs1` offset by a signed immediate value.

Usage

```
s32 $rd, imm($rs1)
```

Operation

```
*(rs+imm)   = rd[31:24];
*(rs+imm+1) = rd[23:16];
*(rs+imm+2) = rd[15:8];
*(rs+imm+3) = rd[7:0];
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

Store least significant 8 bits of `$rd` into the address located in `$rs1` offset by a signed immediate value.

Usage

```
s8 $rd, imm($rs1)
```

Operation

```
*(rs+imm) = rd[7:0];
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

Sets `$rd` as 1 if the signed representation of `$rs1` is greater than than the signed representation of `$rs2`, 0 otherwise.

Usage

```
sgt $rd, $rs1, $rs2
```

Operation

```
if((signed)rs1 > (signed)rs2)
   rd = 1;
else
   rd = 0;
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

Sets `$rd` as 1 if the signed representation of `$rs1` is greater than the signed immediate field, 0 otherwise.

Usage

```
sgti $rd, $rs1, imm
```

Operation

```
if((signed)rs1 > (signed)imm)
   rd = 1;
else
   rd = 0;
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

Sets `$rd` as 1 if the unsigned representation of `$rs1` is greater than the unsigned immediate field, 0 otherwise.

Usage

```
sgtiu $rd, $rs1, imm
```

Operation

```
if((unsigned)rs1 > (unsigned)imm)
   rd = 1;
else
   rd = 0;
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

Sets `$rd` as 1 if the unsigned representation of `$rs1` is greater than the unsigned representation of `$rs2`, 0 otherwise.

Usage

```
sgtu $rd, $rs1, $rs2
```

Operation

```
if((unsigned)rs1 > (unsigned)rs2)
   rd = 1;
else
   rd = 0;
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

Shift the contents of register `$rs1` left by a number of bits specified in `$rs2`, storing the result in `$rd`

Usage

```
sll $rd, $rs1, $rs2
```

Operation

```
rd = rs1<<rs2;
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

Shift the contents of register `$rs1` left by a number of bits specified in the immediate field, storing the result in `$rd`

Usage

```
slli $rd, $rs1, imm
```

Operation

```
rd = rs1<<imm;
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

Sets `$rd` as 1 if the signed representation of `$rs1` is less than the signed representation of `$rs2`, 0 otherwise.

Usage

```
slt $rd, $rs1, $rs2
```

Operation

```
if((signed)rs1 < (signed)rs2)
   rd = 1;
else
   rd = 0;
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

Sets `$rd` as 1 if the signed representation of `$rs1` is less than the signed immediate field, 0 otherwise.

Usage

```
slti $rd, $rs1, imm
```

Operation

```
if((signed)rs1 < (signed)imm)
   rd = 1;
else
   rd = 0;
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

Sets `$rd` as 1 if the unsigned representation of `$rs1` is less than the unsigned immediate field, 0 otherwise.

Usage

```
sltiu $rd, $rs1, imm
```

Operation

```
if((unsigned)rs1 < (unsigned)imm)
   rd = 1;
else
   rd = 0;
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

Sets `$rd` as 1 if the unsigned representation of `$rs1` is less than the unsigned representation of `$rs2`, 0 otherwise.

Usage

```
sltu $rd, $rs1, $rs2
```

Operation

```
if((unsigned)rs1 < (unsigned)rs2)
   rd = 1;
else
   rd = 0;
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

Shift the contents of register `$rs1` right by a number of bits specified in `$rs2`, sign-extending the value from `$rs1` and storing the result in `$rd`

Usage

```
sra $rd, $rs1, $rs2
```

Operation

```
rd = sign_extend(rs1)>>rs2;
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

Shift the contents of register `$rs1` right by a number of bits specified in the immediate field, sign-extending the value from `$rs1` and storing the result in `$rd`

Usage

```
srai $rd, $rs1, imm
```

Operation

```
rd = sign_extend(rs1)>>imm;
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

Shift the contents of register `$rs1` right by a number of bits specified in `$rs2`, storing the result in `$rd`

Usage

```
srl $rd, $rs1, $rs2
```

Operation

```
rd = rs1>>rs2;
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

Shift the contents of register `$rs1` right by a number of bits specified in the immediate field, storing the result in `$rd`

Usage

```
srli $rd, $rs1, imm
```

Operation

```
rd = rs1>>imm;
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

Subtracts register `$rs2` from `$rs1`, placing the result in `$rd`.

Usage

```
sub $rd, $rs1, $rs2
```

Operation

```
rd = rs1 - rs2;
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

Subtracts a sign-extended immediate value from register `$rs1`, placing the result in `$rd`.

Usage

```
subi Rd, Rs1, imm
```

Operation

```
Rd = Rs1 - sign_extend(imm);
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

Store 64-bit word `$rd` into the address located in `$rs1` offset by a signed immediate value.

Usage

```
sw $rd, imm($rs1)
```

Operation

```
*(rs+imm)   = rd[63:56];
*(rs+imm+1) = rd[55:48];
*(rs+imm+2) = rd[47:40];
*(rs+imm+3) = rd[39:32];
*(rs+imm+4) = rd[31:24];
*(rs+imm+5) = rd[23:16];
*(rs+imm+6) = rd[15:8];
*(rs+imm+7) = rd[7:0];
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

Request for kernel service (presently unimplemented pending addition of interrupts).

Usage

```
syscall
```

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

XORs `$rs1` with `$rs2`, placing the result in `$rd`.

Usage

```
xor $rd, $rs1, $rs2
```

Operation

```
rd = rs1 ^ rs2;
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

XORs `$rs1` with an immediate value, placing the result in `$rd`.

Usage

```
xori $rd, $rs1, imm
```

Operation

```
rd = rs1 ^ imm;
advance_pc();
```

Encoding

Type 1

Unit 3

Op 3

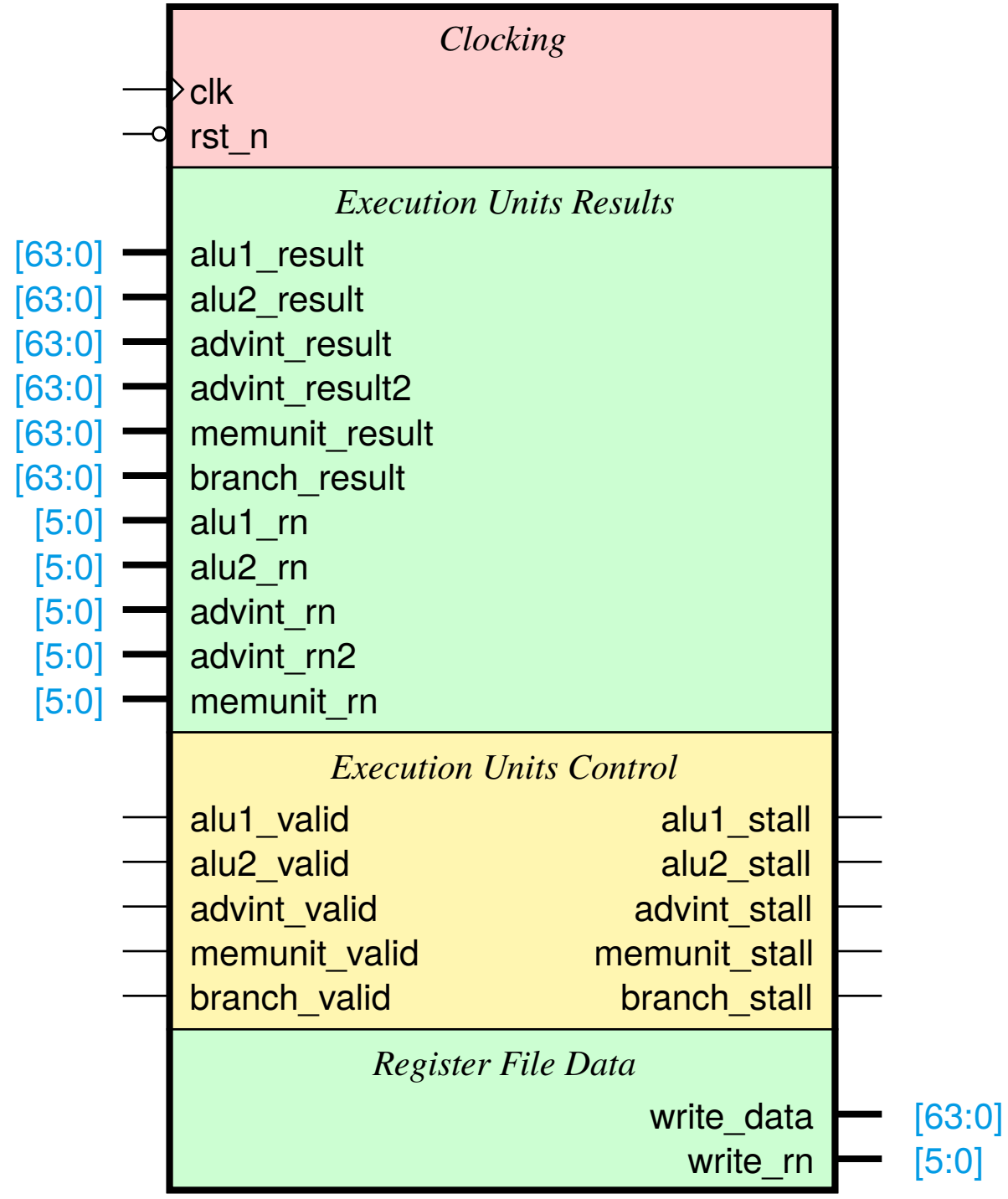
16-bit Opcode NONE

32-bit Opcode 0xAF

64-bit Opcode 0xEF

5.2 Verilog Module Index

5.2.1 commit.v




```
1 module commit(
2     //# {{clocks|Clocking}}
```

(continues on next page)

(continued from previous page)

```
    input clk,
    input rst_n,

    //# {{data|Execution Units Results}}
    input[63:0] alu1_result,
    input[63:0] alu2_result,
    input[63:0] advint_result,
    input[63:0] advint_result2,
    input[63:0] memunit_result,
    input[63:0] branch_result,

    input[5:0] alu1_rn,
    input[5:0] alu2_rn,
    input[5:0] advint_rn,
    input[5:0] advint_rn2,
    input[5:0] memunit_rn,

    //# {{control|Execution Units Control}}
    input alu1_valid,
    input alu2_valid,
    input advint_valid,
    input memunit_valid,
    input branch_valid,

    output alu1_stall,
    output alu2_stall,
    output advint_stall,
    output memunit_stall,
    output branch_stall,

    //# {{data|Register File Data}}
    output[63:0] write_data,
    output[5:0] write_rn
    );

    localparam STATE_IDLE = 3'h0;
    localparam STATE_P1 = 3'h1;
    localparam STATE_P2 = 3'h2;
    localparam STATE_P3 = 3'h3;
    localparam STATE_P4 = 3'h4;
    localparam STATE_P5 = 3'h5;
    localparam STATE_P6 = 3'h6;

    reg[2:0] state, next_state;

    reg[5:0] pending_rn[1:6];
    reg[63:0] pending_data[1:6];
    reg[6:1] pending_valid;

    //TODO Finish exception handling by un-stalling the ex units in strict order of␣
↪issue (via external counting table)
    //For now, no exceptions are implemented.
    assign alu1_stall = pending_valid[1] && state != STATE_P1;
    assign alu2_stall = pending_valid[2] && state != STATE_P2;
    assign advint_stall = (pending_valid[3] && state != STATE_P3) | (pending_valid[4]␣
↪&& state != STATE_P4);
    assign memunit_stall = pending_valid[5] && state != STATE_P5;
```

(continues on next page)

(continued from previous page)

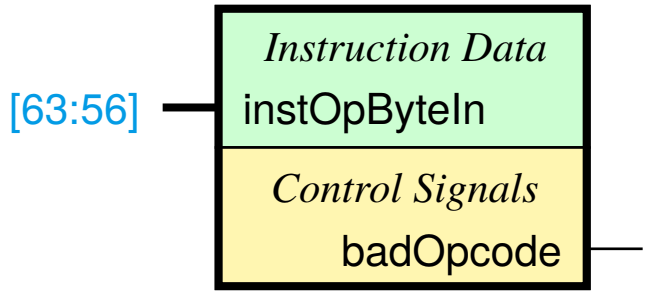
```
58      assign branch_stall = pending_valid[6] && state != STATE_P6;
59
60      integer i;
61
62
63      //TODO A side-effect of the current implemtation is that a short running␣
    ↪instruction issued after a long running
64      //one which affects the same destination register might get written/overwritten␣
    ↪in the wrong order. This will be
65      //fixed by the proposed PC instruction graduation control table necessary for␣
    ↪exceptions
66
67      //The *unit*_valid signals need to remain high for only one cycle per transaction␣
    ↪or all hell will break loose.
68      always @(posedge clk or negedge rst_n)
69      begin
70          if(~rst_n) begin
71              for(i = 1; i <= 6; i = i + 1) begin
72                  pending_data[i] <= 64'h0;
73                  pending_rn[i] <= 6'h0;
74                  pending_valid[i] <= 0;
75              end
76          end else begin
77              case(state)
78              STATE_P1: pending_valid[1] <= 0;
79              STATE_P2: pending_valid[2] <= 0;
80              STATE_P3: pending_valid[3] <= 0;
81              STATE_P4: pending_valid[4] <= 0;
82              STATE_P5: pending_valid[5] <= 0;
83              STATE_P6: pending_valid[6] <= 0;
84              endcase
85
86              if(alu1_valid & |alu1_rn) begin
87                  pending_data[1] <= alu1_result;
88                  pending_rn[1] <= alu1_rn;
89                  pending_valid[1] <= 1;
90
91              end if(alu2_valid & |alu2_rn) begin
92                  pending_data[2] <= alu2_result;
93                  pending_rn[2] <= alu2_rn;
94                  pending_valid[2] <= 1;
95
96              end if(advint_valid & (|advint_rn | |advint_rn2)) begin
97                  pending_data[3] <= advint_result;
98                  pending_rn[3] <= advint_rn;
99                  pending_valid[3] <= 1;
100                 pending_data[4] <= advint_result2;
101                 pending_rn[4] <= advint_rn2;
102                 pending_valid[4] <= 1;
103
104             end if(memunit_valid & |memunit_rn) begin
105                 pending_data[5] <= memunit_result;
106                 pending_rn[5] <= memunit_rn;
107                 pending_valid[5] <= 1;
108
109             end if(branch_valid) begin
110                 pending_data[6] <= branch_result;
```

(continues on next page)

(continued from previous page)

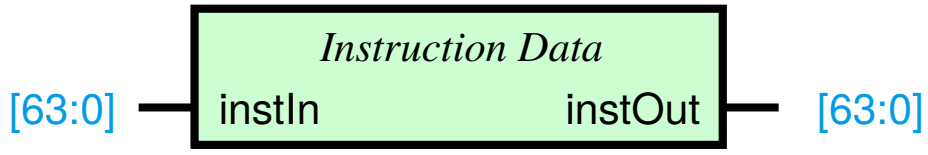
```
                pending_rn[6] <= 6'd63;
                pending_valid[6] <= 1;
            end
        end
    end

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) state <= STATE_IDLE;
        else state <= next_state;
    end

    always @(*)
    begin
        if((pending_valid[6] & state != STATE_P6) | branch_valid) next_state = STATE_P6;
        else if((pending_valid[5] & state != STATE_P5) | (memunit_valid & |memunit_rn)) next_state = STATE_P5;
        else if((pending_valid[4] & state != STATE_P4) | (advint_valid & |advint_rn2)) next_state = STATE_P4;
        else if((pending_valid[3] & state != STATE_P3) | (advint_valid & |advint_rn)) next_state = STATE_P3;
        else if((pending_valid[2] & state != STATE_P2) | (alu2_valid & |alu2_rn)) next_state = STATE_P2;
        else if((pending_valid[1] & state != STATE_P1) | (alu1_valid & |alu1_rn)) next_state = STATE_P1;
        else next_state = STATE_IDLE;
    end

    assign write_data = state == STATE_P1 ? pending_data[1] :
                        state == STATE_P2 ? pending_data[2] :
                        state == STATE_P3 ? pending_data[3] :
                        state == STATE_P4 ? pending_data[4] :
                        state == STATE_P5 ? pending_data[5] :
                        state == STATE_P6 ? pending_data[6] :
                        64'h0;

    assign write_rn = state == STATE_P1 ? pending_rn[1] :
                      state == STATE_P2 ? pending_rn[2] :
                      state == STATE_P3 ? pending_rn[3] :
                      state == STATE_P4 ? pending_rn[4] :
                      state == STATE_P5 ? pending_rn[5] :
                      state == STATE_P6 ? pending_rn[6] :
                      6'h0;

endmodule
```

5.2.2 de_badDetect.v




```
//Raisin64 Decode Unit - Bad Opcode Detector

module de_badDetect(
    //# {{data|Instruction Data}}
    input[63:56] instOpByteIn,

    //# {{control|Control Signals}}
    output badOpcode
    );

    `include "de_isa_def.vh"

    reg badOpcode_pre;
    wire is16, is32, is64;

    assign badOpcode = badOpcode_pre;

    assign is16 = ~instOpByteIn[63];
    assign is32 = instOpByteIn[63:62] == 2'h2;
    assign is64 = instOpByteIn[63:62] == 2'h3;

    //Detects and flags invalid opcodes
    always @(*)
    begin
        badOpcode_pre = 0;

        if(is16 && instOpByteIn[62:60] == 3'h7) badOpcode_pre = 1;
        else if(is32 | is64)
        begin
            case(instOpByteIn[61:56])
                `OP_BAD_02, `OP_BAD_03, `OP_BAD_0B, `OP_BAD_14, `OP_BAD_15,
                `OP_BAD_16, `OP_BAD_17, `OP_BAD_18, `OP_BAD_19, `OP_BAD_1A,
                `OP_BAD_1B, `OP_BAD_22, `OP_BAD_23, `OP_BAD_2B: badOpcode_pre = 1;
                `OP_FSTAR, `OP_LUI, `OP_JALI, `OP_JI: if(is32) badOpcode_pre = 1;
            endcase
        end
    end
endmodule
```

5.2.3 de_canonicalize.v



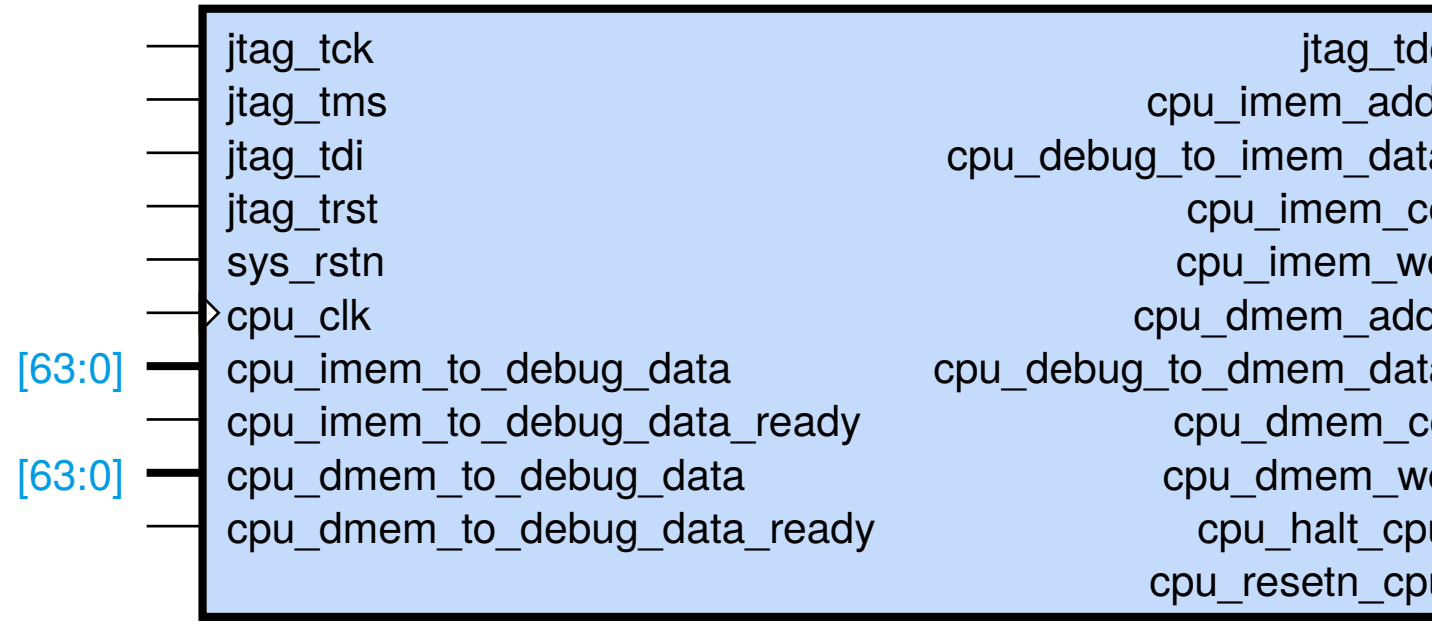

```
//Raisin64 Decode Unit - Opcode Canonicalization
//Converts compact instructions into their true 64-bit native format

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
        end

        //32-bit instructions
```

(continues on next page)

(continued from previous page)

```
        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
```

5.2.4 debug_control.v




```
/* debug_control.v
 *
 * Example Debug Controller for a simple CPU using the JTAGlet interface
 *
 *-------------------------------------------------------------------------------
 *
 * Copyright 2018 Christopher Parish
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

module debug_control(
    input jtag_tck,
    input jtag_tms,
    input jtag_tdi,
    input jtag_trst,
    output jtag_tdo,

    input sys_rstn, //System reset. Should NOT be externally tied to our cpu_resetn_
↪cpu output

    input cpu_clk,
```

(continues on next page)

(continued from previous page)

```
32
33      output reg[63:0] cpu_imem_addr,
34      output reg[63:0] cpu_debug_to_imem_data,
35      input[63:0] cpu_imem_to_debug_data,
36      input cpu_imem_to_debug_data_ready,
37      output reg cpu_imem_ce,
38      output reg cpu_imem_we,
39
40      output reg[63:0] cpu_dmem_addr,
41      output reg[63:0] cpu_debug_to_dmem_data,
42      input[63:0] cpu_dmem_to_debug_data,
43      input cpu_dmem_to_debug_data_ready,
44      output reg cpu_dmem_ce,
45      output reg cpu_dmem_we,
46
47      output reg cpu_halt_cpu,
48      output cpu_resetn_cpu
49      );
50
51      //Signals from the JTAG TAP to the synchronizer
52      wire jtag_userOp_ready;
53
54      //Resulting signal in the CPU domain
55      wire cpu_userOp_ready;
56
57      //Requested operation/data from the TAP in the JTAG domain
58      wire[7:0] jtag_userOp;
59      wire[63:0] jtag_userData;
60
61      reg[63:0] cpu_userData;
62
63      //The Jtaglet JTAG TAP
64      jtaglet #(.ID_PARTVER(4'h1), .ID_PARTNUM(16'hCAFE), .ID_MANF(11'h035), .USERDATA_
    ↪LEN(64)) jtag_if
65          (.tck(jtag_tck), .tms(jtag_tms), .tdo(jtag_tdo), .tdi(jtag_tdi), .trst(jtag_
    ↪trst),
66           .userData_out(jtag_userData), .userData_in(cpu_userData), .userOp(jtag_
    ↪userOp),
67           .userOp_ready(jtag_userOp_ready));
68
69      //Synchronizer to take the userOp ready signal into the CPU clock domain
70      ff_sync #(.WIDTH(1)) userOpReady_toCPUDomain
71          (.clk(cpu_clk), .rst_p(~sys_rstn), .in_async(jtag_userOp_ready), .out(cpu_
    ↪userOp_ready));
72
73      //Stateless debug operations (which ignore debug register contents)
74      localparam DEBUGOP_NOOP_OP      = 8'h00;
75      localparam DEBUGOP_CPUHALT_OP   = 8'h01;
76      localparam DEBUGOP_CPURESUME_OP = 8'h02;
77      localparam DEBUGOP_CPURESET_OP  = 8'h03;
78
79      //Debug operations (use previously stored data to carry out an operation)
80      localparam DEBUGOP_READIMEM_OP  = 8'h04;
81      localparam DEBUGOP_WRITEIMEM_OP = 8'h05;
82      localparam DEBUGOP_READDMEM_OP  = 8'h06;
83      localparam DEBUGOP_WRITEDMEM_OP = 8'h07;
84
```

(continues on next page)

(continued from previous page)

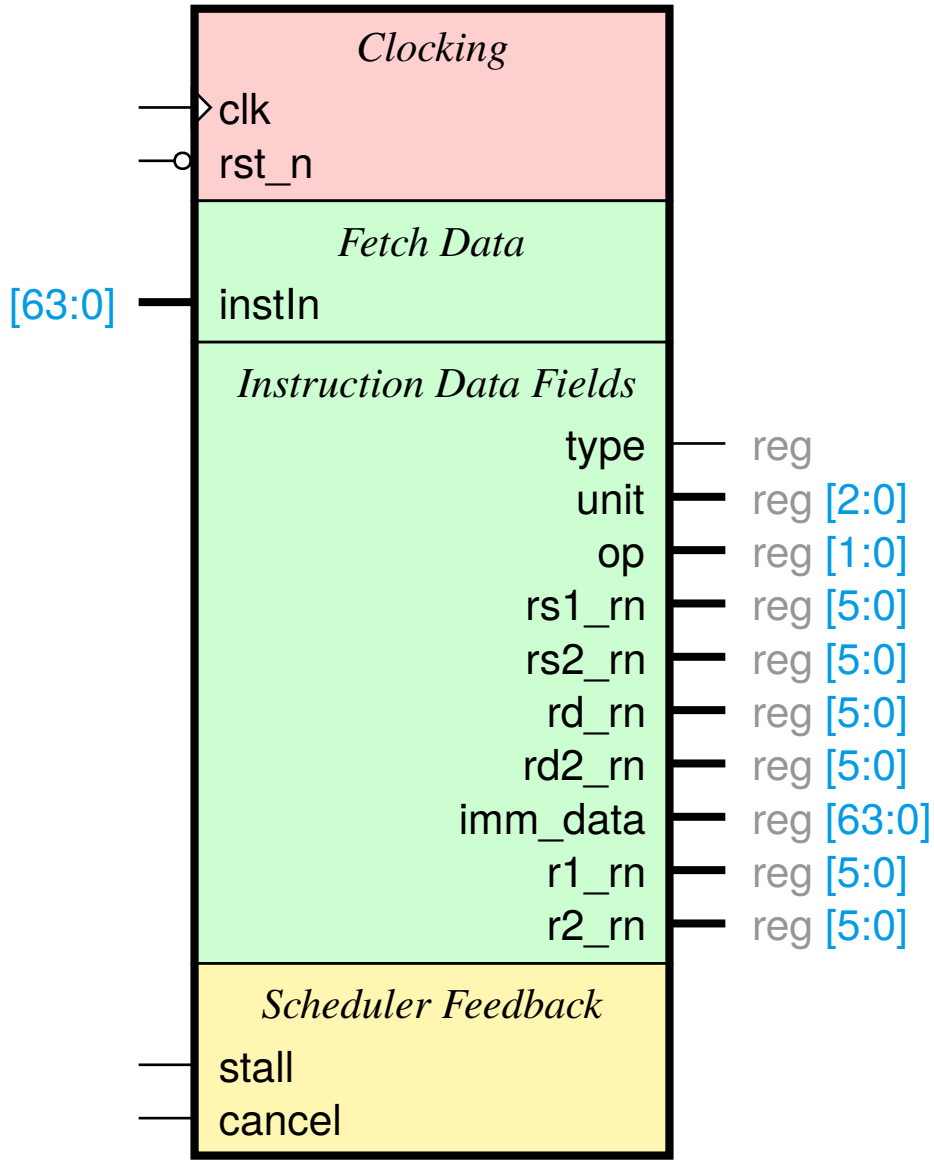
```
85      //Load/store debug operations (have no side-effects apart from
86      //loading/storing the appropriate debug register)
87      localparam DEBUGOP_IADDR_REG     = 8'h80;
88      localparam DEBUGOP_IDATA_REG     = 8'h81;
89      localparam DEBUGOP_DADDR_REG     = 8'h82;
90      localparam DEBUGOP_DDATA_REG     = 8'h83;
91      localparam DEBUGOP_CPUFLAGS_REG  = 8'h84;
92
93      reg cpu_userOp_ready_last;
94      wire execUserOp = ~cpu_userOp_ready_last & cpu_userOp_ready;
95
96      always @(posedge cpu_clk or negedge sys_rstn) begin
97          if(~sys_rstn) cpu_userOp_ready_last <= 0;
98          else cpu_userOp_ready_last <= cpu_userOp_ready;
99      end
100
101     always @(posedge cpu_clk or negedge sys_rstn) begin
102         if(~sys_rstn) begin
103             cpu_imem_we <= 0;
104             cpu_imem_ce <= 0;
105         end else begin
106             cpu_imem_we <= 0;
107             cpu_imem_ce <= 0;
108             if(execUserOp) case(jtag_userOp)
109                 DEBUGOP_READIMEM_OP: cpu_imem_ce <= 1;
110                 DEBUGOP_WRITEIMEM_OP: begin
111                     cpu_imem_we <= 1;
112                     cpu_imem_ce <= 1;
113                 end
114             endcase
115         end
116     end
117
118     always @(posedge cpu_clk or negedge sys_rstn) begin
119         if(~sys_rstn) begin
120             cpu_dmem_we <= 0;
121             cpu_dmem_ce <= 0;
122         end else begin
123             cpu_dmem_we <= 0;
124             cpu_dmem_ce <= 0;
125             if(execUserOp) case(jtag_userOp)
126                 DEBUGOP_READDMEM_OP: cpu_dmem_ce <= 1;
127                 DEBUGOP_WRITEDMEM_OP: begin
128                     cpu_dmem_we <= 1;
129                     cpu_dmem_ce <= 1;
130                 end
131             endcase
132         end
133     end
134
135     always @(posedge cpu_clk) begin
136         if(execUserOp) case(jtag_userOp)
137             DEBUGOP_IADDR_REG: cpu_imem_addr <= jtag_userData;
138         endcase
139     end
140
141     always @(posedge cpu_clk) begin
```

(continues on next page)

(continued from previous page)

```
        if(execUserOp) case(jtag_userOp)
            DEBUGOP_IDATA_REG: cpu_debug_to_imem_data <= jtag_userData;
        endcase
    end

    always @(posedge cpu_clk) begin
        if(execUserOp) case(jtag_userOp)
            DEBUGOP_DADDR_REG: cpu_dmem_addr <= jtag_userData;
        endcase
    end

    always @(posedge cpu_clk) begin
        if(execUserOp) case(jtag_userOp)
            DEBUGOP_DDATA_REG: cpu_debug_to_dmem_data <= jtag_userData;
        endcase
    end

    always @(posedge cpu_clk or negedge sys_rstn) begin
        if(~sys_rstn) begin
            cpu_userData <= 0;
        end else begin
            if(cpu_imem_to_debug_data_ready) cpu_userData <= cpu_imem_to_debug_data;
            else if(cpu_dmem_to_debug_data_ready) cpu_userData <= cpu_dmem_to_debug_
↪data;
        end
    end

    //Reset Stretcher
    reg requestReset;
    reg[9:0] resetStretch;
    assign cpu_resetn_cpu = ~(|resetStretch);
    always @(posedge cpu_clk or negedge sys_rstn) begin
        if(~sys_rstn) resetStretch <= 10'b0;
        else if(requestReset) resetStretch <= {10{1'b1}};
        else if(resetStretch != 0) resetStretch <= resetStretch - 1;
    end

    always @(posedge cpu_clk or negedge sys_rstn) begin
        if(~sys_rstn) begin
            cpu_halt_cpu <= 0;
            requestReset <= 0;
        end else begin
            requestReset <= 0;
            if(execUserOp) case(jtag_userOp)
                DEBUGOP_CPUHALT_OP: cpu_halt_cpu <= 1;
                DEBUGOP_CPURESUME_OP: cpu_halt_cpu <= 0;
                DEBUGOP_CPURESET_OP: begin
                    cpu_halt_cpu <= 0;
                    requestReset <= 1;
                end
            endcase
        end
    end

endmodule
```

5.2.5 decode.v




```
1  //Raisin64 Instruction Decode
2
3  module decode(
4      //# {{clocks|Clocking}}
5      input clk,
6      input rst_n,
7
8      //# {{data|Fetch Data}}
9      input[63:0] instIn,
10
11     //# {{data|Instruction Data Fields}}
12     output reg type,
13     output reg[2:0] unit,
14     output reg[1:0] op,
```

(continues on next page)

(continued from previous page)

```
    output reg[5:0] rs1_rn,
    output reg[5:0] rs2_rn,
    output reg[5:0] rd_rn,
    output reg[5:0] rd2_rn,
    output reg[63:0] imm_data,

    //Indicates which registers are loaded for this instruction
    output reg[5:0] r1_rn,
    output reg[5:0] r2_rn,

    //# {{control|Scheduler Feedback}}
    input stall,
    input cancel
    );

    wire[63:0] canonInst;
    wire badOpcode;

    de_canonicalize de_canonicalize_1(
        .instIn(instIn), .instOut(canonInst)
        );

    de_badDetect de_badDetect_1(
        .instOpByteIn(instIn[63:56]), .badOpcode(badOpcode)
        );

    reg load_rs1, load_rs1_rs2, load_rs1_rd;

    reg signedImm;

    always @(*) begin
        signedImm = 0;
        casex(canonInst[60:56])
        5'b000xx, //ADD, SUB
        5'b001x0, //SLTI, SGTI
        5'b10xxx, //LW, L32, L16, L8, LUI, L32S, L16S, L8S
        5'b110xx, //SW, S32, S16, S8
        5'b1110x: signedImm = 1; //BEQ, BEQAL
        endcase
    end

    wire ji_type;
    assign ji_type = &canonInst[61:57]; //Imm Type, Unit 7, JALI or JI

    always @(*) begin
        load_rs1 = 0;
        load_rs1_rs2 = 0;
        load_rs1_rd = 0;

        if(~canonInst[61]) begin //R-Type
            if(canonInst[60:58] < 3'h5 || //Units 0-4
              (&canonInst[60:58] && canonInst[57:56] == 2'h1)) begin //F* Inst
                load_rs1_rs2 = 1;
            end else if(&canonInst[60:58] & canonInst[57]) begin //JAL, J
                load_rs1 = 1;
            end
        end else begin //I-Type
```

(continues on next page)

(continued from previous page)

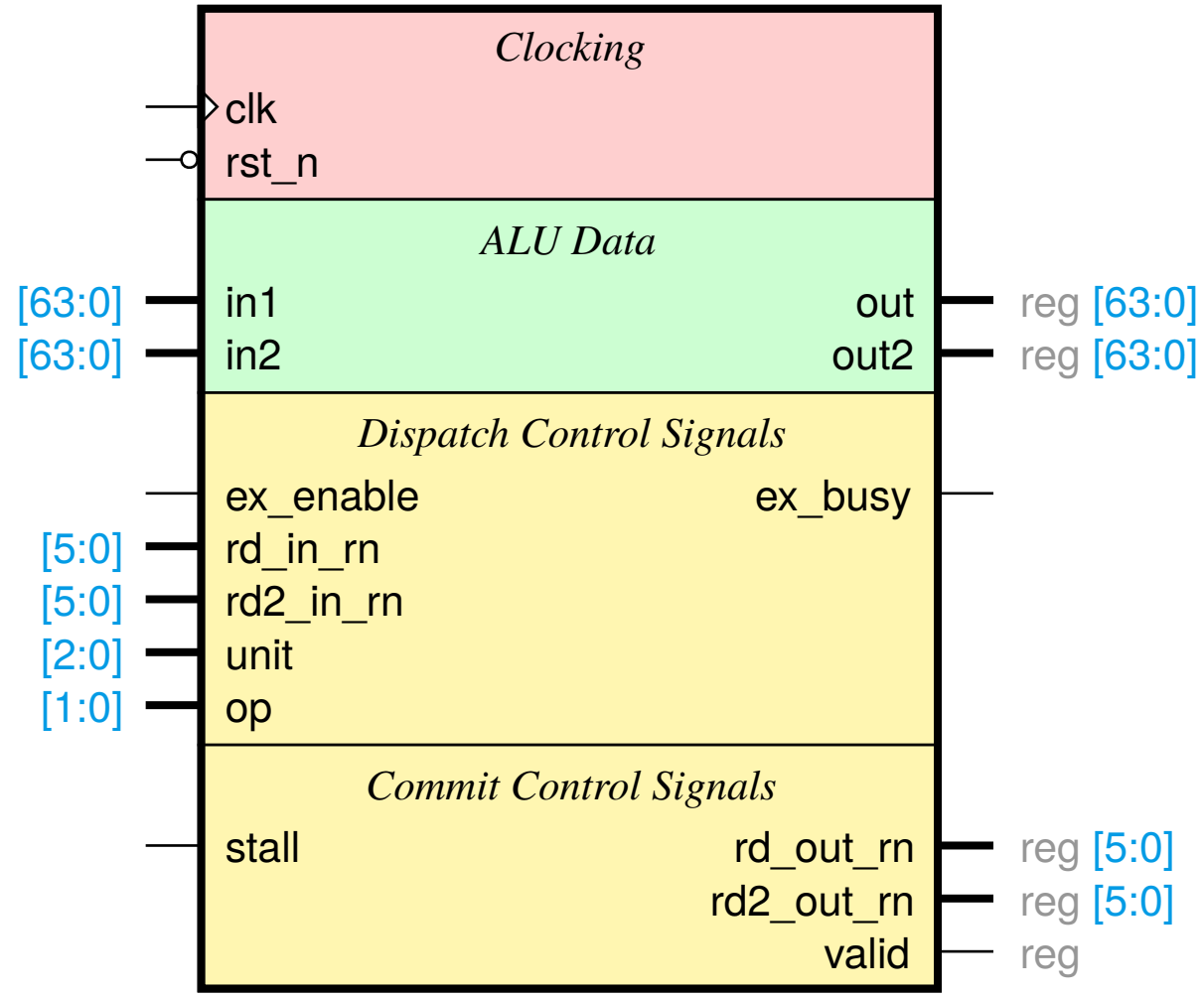
```
            if(canonInst[60:58] < 3'h5 || //Units 0-4
              canonInst[60:58] == 3'h5 && |canonInst[57:56]) begin //Unit 5 except LUI
                load_rs1 = 1;
            end else if(canonInst[60:58] == 3'h6 | //Unit 6
              &canonInst[60:58] & ~canonInst[57]) begin //BEQ, BEQAL
                load_rs1_rd = 1;
            end
        end
    end

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            type <= 0;
            unit <= 0;
            op <= 0;
            rs1_rn <= 0;
            rs2_rn <= 0;
            rd_rn <= 0;
            rd2_rn <= 0;
            imm_data <= 0;
            r1_rn <= 0;
            r2_rn <= 0;
        end else begin
            if(cancel) begin
                type <= 0;
                unit <= 0;
                op <= 0;
                rs1_rn <= 0;
                rs2_rn <= 0;
                rd_rn <= 0;
                rd2_rn <= 0;
                imm_data <= 0;
                r1_rn <= 0;
                r2_rn <= 0;
            end else if(~stall) begin
                type <= canonInst[61];
                unit <= canonInst[60:58];
                op <= canonInst[57:56];
                rs1_rn <= canonInst[43:38];
                rs2_rn <= canonInst[37:32];
                rd_rn <= canonInst[55:50];
                rd2_rn <= canonInst[49:44];
                imm_data <= ji_type ? {{7{1'b0}},canonInst[55:0],1'b0} : //TODO Need
↪to decide how upper bits are handled
                            signedImm ? {{32{canonInst[31]}},canonInst[31:0]} :
                            {{32{1'b0}},canonInst[31:0]};

                r1_rn <= (load_rs1|load_rs1_rs2|load_rs1_rd) ? canonInst[43:38] : 6
↪'h0;
                r2_rn <= load_rs1_rd ? canonInst[55:50] :
                                       load_rs1_rs2 ? canonInst[37:32] :
                                       6'h0;
            end
        end
    end

```

(continues on next page)

(continued from previous page)

```
127 endmodule
```

5.2.6 ex_advint.v



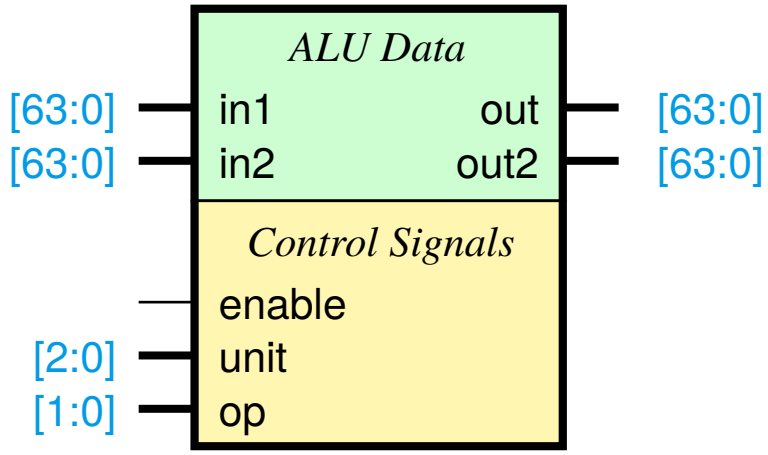

```
1  //Raisin64 Execute Unit - Advanced Integer Unit
2
3  module ex_advint(
4      //# {{clocks|Clocking}}
5      input clk,
6      input rst_n,
7
8      //# {{data|ALU Data}}
9      input[63:0] in1,
10     input[63:0] in2,
11     output reg[63:0] out,
12     output reg[63:0] out2,
13
14     //# {{control|Dispatch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

(continues on next page)

(continued from previous page)

```
    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
```

5.2.7 ex_advint_s1.v



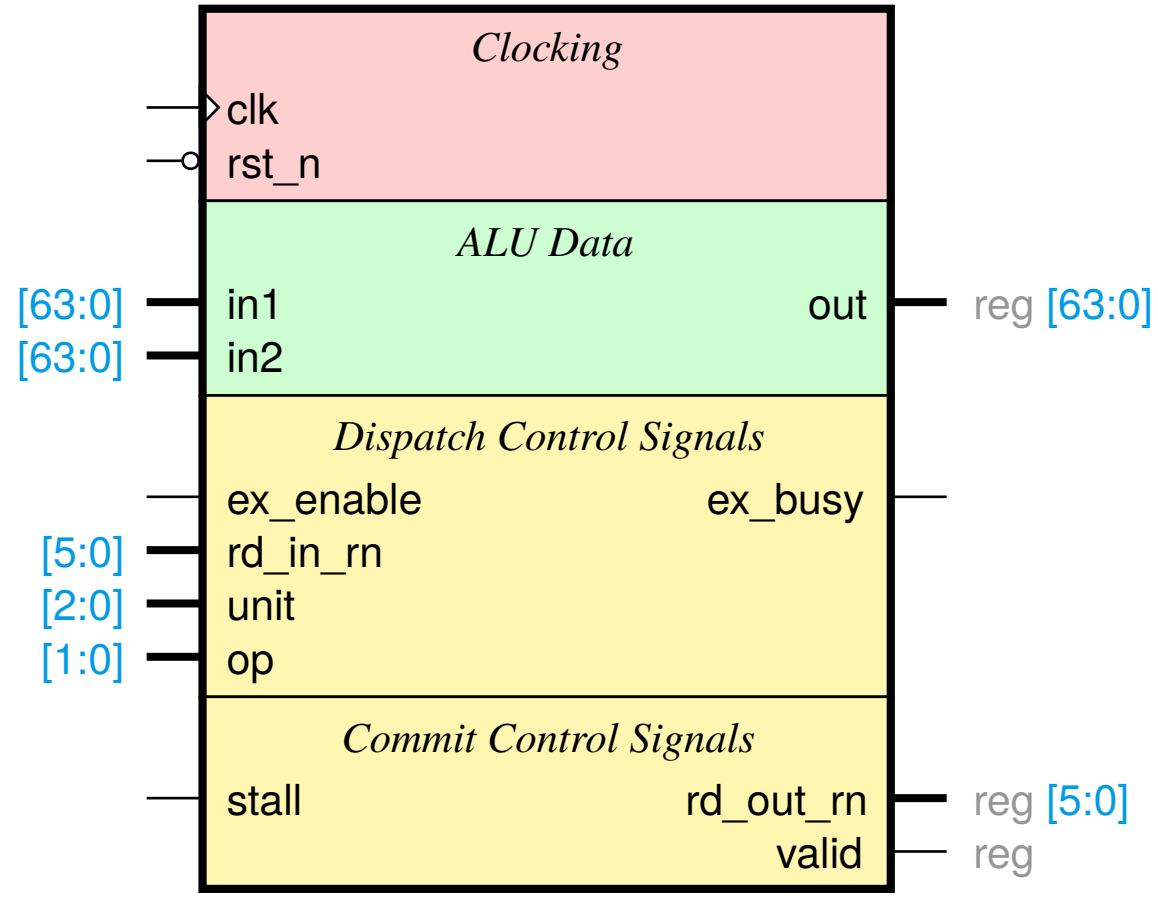

```
//Raisin64 Execute Unit - Advanced Integer Unit - Stage 1

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

module ex_advint_s1(
    //# {{data|ALU Data}}
    input[63:0] in1,
    input[63:0] in2,
    output[63:0] out,
    output[63:0] out2,

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

    reg[127:0] out_pre;
    assign out = out_pre[63:0];
    assign out2 = out_pre[127:64];

    always @(*)
    begin
        out_pre = 128'h0;

        if(enable & unit==3'h4) begin
            case(op)
                0: out_pre = $signed(in1) * $signed(in2); //MUL
                1: out_pre = in1 * in2; //MULU
                2,3: out_pre = {in2, in1}; //TODO div stubbed for now
//              2: begin //DIV
//                  out_pre[63:0] = $signed(in1) / $signed(in2);
//                  out_pre[127:64] = $signed(in1) % $signed(in2);
//                 end
//              3: begin //DIVU
//                  out_pre[63:0] = in1 / in2;
//                  out_pre[127:64] = in1 % in2;
```

(continues on next page)

(continued from previous page)

```
39 //                  end
40             endcase
41         end
42     end
43 endmodule
```

5.2.8 ex_alu.v



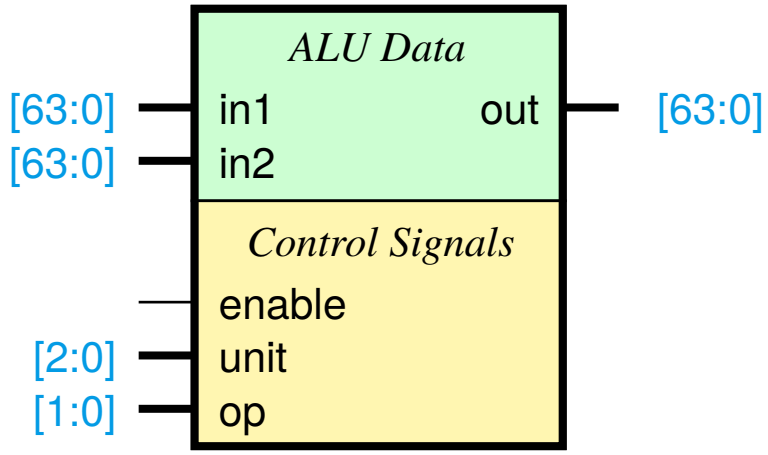

```
1  //Raisin64 Execute Unit - Integer ALU
2
3  module ex_alu(
4      //# {{clocks|Clocking}}
5      input clk,
6      input rst_n,
7
8      //# {{data|ALU Data}}
9      input[63:0] in1,
10     input[63:0] in2,
11     output reg[63:0] out,
12
13     //# {{control|Dispatch Control Signals}}
14     input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

(continues on next page)

(continued from previous page)

```
    );

    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
```

5.2.9 ex_alu_s1.v



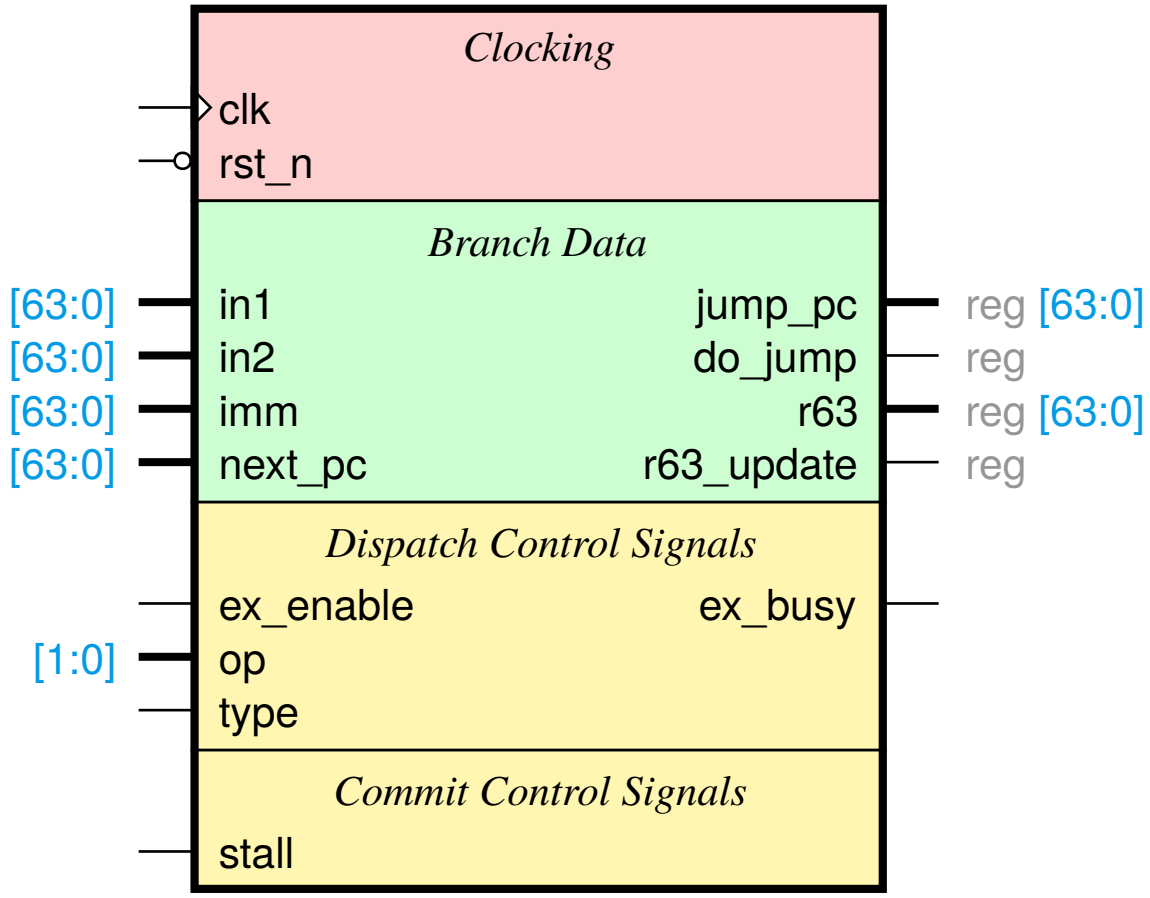

```
//Raisin64 Execute Unit - Integer ALU Stage 1

module ex_alu_s1(
    //# {{data|ALU Data}}
    input[63:0] in1,
    input[63:0] in2,
    output[63:0] out,

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

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

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

(continues on next page)

(continued from previous page)

```
                endcase
            3'h3: //Bitwise Operations
                case(op)
                    0: out_pre = in1 & in2; //AND
                    1: out_pre = !(in1 | in2); //NOR
                    2: out_pre = in1 | in2; //OR
                    3: out_pre = in1 ^ in2; //XOR
                endcase
        endcase
    end
endmodule
```

5.2.10 ex_branch.v



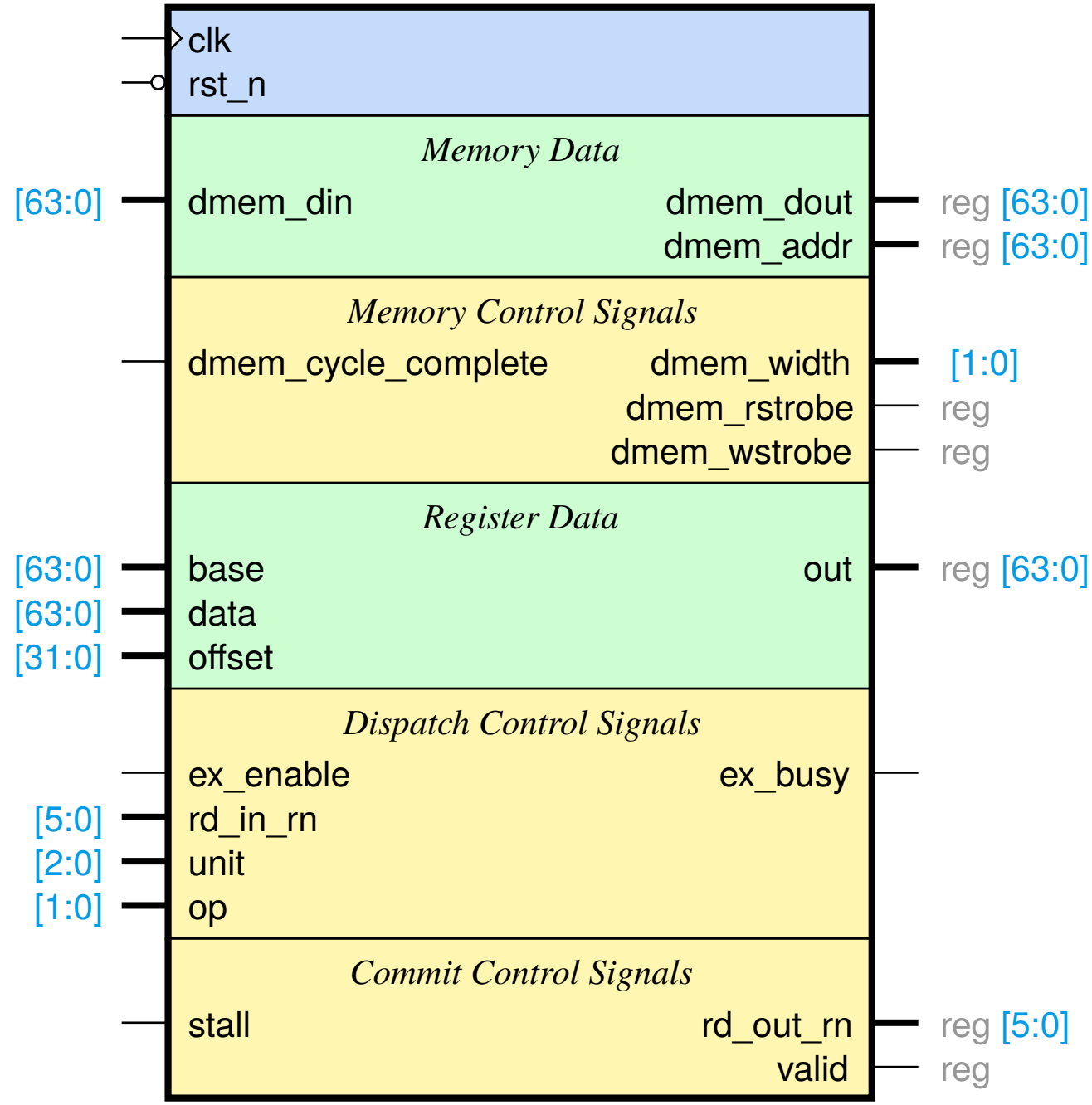

```
1  //Raisin64 Execute Unit - Branch and Jump Unit
2
3  module ex_branch(
4      //# {{clocks|Clocking}}
5      input clk,
6      input rst_n,
7
8      //# {{data|Branch Data}}
9      input[63:0] in1,
10     input[63:0] in2,
11     input[63:0] imm,
12     input[63:0] next_pc,
13     output reg[63:0] jump_pc,
14     output reg do_jump,
15
16     output reg[63:0] r63,
17     output reg r63_update,
18
19     //# {{control|Dispatch Control Signals}}
20     input ex_enable,
21     output ex_busy,
22     input[1:0] op,
23     input type,
```

(continues on next page)

(continued from previous page)

```
24
25      //# {{control|Commit Control Signals}}
26      input stall
27      );
28
29      wire op_eq;
30      assign op_eq = (in1 == in2);
31
32      always @(posedge clk or negedge rst_n)
33      begin
34          if(~rst_n) begin
35              jump_pc <= 64'h0;
36              do_jump <= 0;
37              r63 <= 64'h0;
38              r63_update <= 0;
39          end else begin
40              jump_pc <= 64'h0;
41              do_jump <= 0;
42              r63 <= 64'h0;
43              r63_update <= 0;
44
45              if(ex_enable) begin
46                  if(op[1]) begin //Jump
47                      //Input is already properly shifted by the time it gets here
48                      jump_pc <= type ? imm : in1;
49                      do_jump <= 1;
50                      if(op[0]) begin //And Link
51                          r63_update <= 1;
52                          r63 <= next_pc;
53                      end
54                  end else if (~op[1] & op_eq) begin //Branch
55                      jump_pc <= next_pc + (imm<<1);
56                      do_jump <= 1;
57                      if(op[0]) begin //And Link
58                          r63_update <= 1;
59                          r63 <= next_pc;
60                      end
61                  end
62              end
63          end
64      end
65
66      assign ex_busy = ex_enable | stall;
67
68  endmodule
```

5.2.11 ex_memory.v




```
1  //Raisin64 Execute Unit - Memory Unit
2
3  `include "io_def.vh"
4
5  module ex_memory(
6      input clk,
7      input rst_n,
8
```

(continues on next page)

(continued from previous page)

```
    //# {{data|Memory Data}}
    input[63:0] dmem_din,
    output reg[63:0] dmem_dout,
    output reg[63:0] dmem_addr,

    //# {{control|Memory Control Signals}}
    input dmem_cycle_complete,
    output[1:0] dmem_width,
    output reg dmem_rstrobe,
    output reg dmem_wstrobe,

    //# {{data|Register Data}}
    input[63:0] base, //R1
    input[63:0] data, //R2
    input[31:0] offset,
    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
    );

    localparam START = 2'h0;
    localparam READ_WAIT = 2'h1;
    localparam WRITE_WAIT = 2'h2;

    ////////////////////////////////////////
    //// Registering of control signals ////
    ////////////////////////////////////////

    reg[5:0] rd_in_rn_reg;
    reg[2:0] unit_reg;
    reg[1:0] op_reg;

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            rd_in_rn_reg <= 6'h0;
            unit_reg <= 3'h0;
            op_reg <= 2'h0;
        end else if(ex_enable) begin
            rd_in_rn_reg <= rd_in_rn;
            unit_reg <= unit;
            op_reg <= op;
        end
    end

    //Signals associated with the incoming instruction (and only valid at ex_enable)
    wire load, store, lui;
```

(continues on next page)

(continued from previous page)

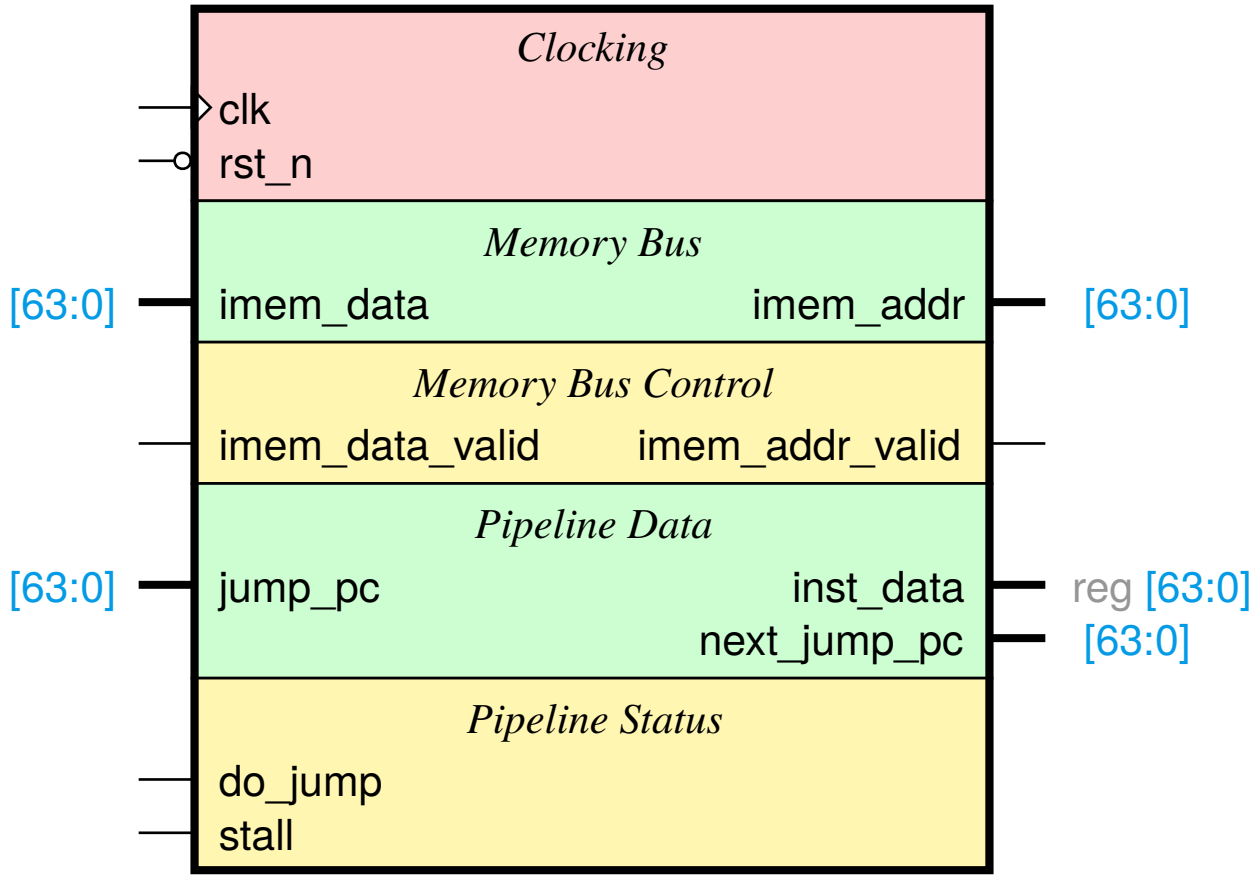
```
66      assign load = (unit == 3'h4 || //Regular load
67                     (unit == 3'h5 && op != 2'h0)); //Sign-Extend load except LUI
68
69      assign store = (unit == 3'h6); //Store
70      assign lui = (unit == 3'h5 && op == 2'h0); //LUI
71
72      //Signals using the registered control signals
73      wire sign_extend_reg;
74      assign sign_extend_reg = (unit_reg == 3'h5 && op_reg != 2'h0); //Sign-Extend load␣
    ↪except LUI
75
76      assign dmem_width = op_reg;
77
78      ////////////////////////////////
79      //// State machine control ////
80      ////////////////////////////////
81
82      reg[1:0] state;
83
84      assign ex_busy = ex_enable || stall || (state != START && ~dmem_cycle_complete);
85
86      always @(posedge clk or negedge rst_n)
87      begin
88          if(~rst_n) begin
89              state <= START;
90              out <= 64'h0;
91              valid <= 1'b0;
92              rd_out_rn <= 6'h0;
93              dmem_dout <= 64'h0;
94              dmem_addr <= 64'h0;
95              dmem_rstrobe <= 1'b0;
96              dmem_wstrobe <= 1'b0;
97          end else begin
98
99              case(state)
100             START: begin
101                 valid <= 0;
102                 rd_out_rn <= 6'h0;
103                 dmem_rstrobe <= 0;
104                 dmem_wstrobe <= 0;
105
106                 if(ex_enable) begin
107                     if(load) begin
108                         state <= READ_WAIT;
109                         dmem_addr <= base + offset;
110                         dmem_rstrobe <= 1;
111                     end else if(store) begin
112                         dmem_addr <= base + offset;
113                         dmem_dout <= data;
114                         dmem_wstrobe <= 1;
115                         state <= WRITE_WAIT;
116                     end else if(lui) begin
117                         out <= {offset,{32{1'b0}}};
118                         valid <= 1;
119                         rd_out_rn <= rd_in_rn;
120                     end
121                 end
```

(continues on next page)

(continued from previous page)

```
            end

            READ_WAIT: begin
                dmem_rstrobe <= 0;
                if(dmem_cycle_complete) begin
                    valid <= 1;
                    rd_out_rn <= rd_in_rn_reg;
                    case(op_reg)
                    //64-Bit load
                    `RAM_WIDTH64: out <= dmem_din;

                    //32-Bit load
                    `RAM_WIDTH32: out <= sign_extend_reg ? {{32{dmem_din[63]}},dmem_
↪din[63:32]} :
                                                {{32{1'b0}},dmem_din[63:32]};

                    //16-Bit load
                    `RAM_WIDTH16: out <= sign_extend_reg ? {{48{dmem_din[63]}},dmem_
↪din[63:48]} :
                                                {{48{1'b0}},dmem_din[63:48]};

                    //8-Bit load
                    `RAM_WIDTH8: out <= sign_extend_reg ? {{56{dmem_din[63]}},dmem_
↪din[63:56]} :
                                                {{56{1'b0}},dmem_din[63:56]};
                    endcase
                    state <= START;
                end
            end

            WRITE_WAIT: begin
                dmem_wstrobe <= 0;
                if(dmem_cycle_complete) begin
                    valid <= 1;
                    state <= START;
                end
            end

            default: begin
                state <= START; //TODO Throw exception
            end
            endcase
        end
    end
endmodule
```

5.2.12 fetch.v




```
1  //Raisin64 - Fetch Module
2  module fetch(
3      //# {{clocks|Clocking}}
4      input clk,
5      input rst_n,
6
7      //# {{data|Memory Bus}}
8      output[63:0] imem_addr,
9      input[63:0] imem_data,
10
11     //# {{control|Memory Bus Control}}
12     input imem_data_valid,
13     output imem_addr_valid,
14
15     //# {{data|Pipeline Data}}
16     output reg[63:0] inst_data,
17     output[63:0] next_jump_pc,
18     input[63:0] jump_pc,
19
20     //# {{control|Pipeline Status}}
21     input do_jump,
22     input stall
23     );
24
```

(continues on next page)

(continued from previous page)

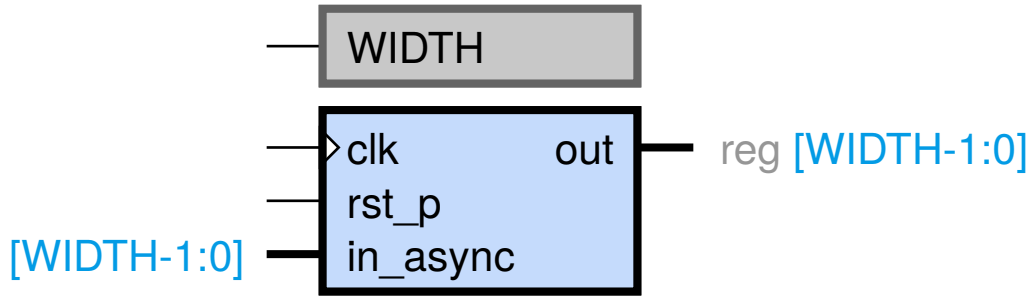
```
    reg[63:0] pc, next_pc, prev_pc;

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            //pc <= 64'h0;
            prev_pc <= 64'h0;
        end else begin
            //pc <= next_pc;
            prev_pc <= pc;
        end
    end

    assign imem_addr = pc;

    //Becuase the PC is "ahead" by a cycle and the data leaving the fetch
    //module is "behind", our prev_pc will actually point at the next PC
    //for any associated jumps.
    assign next_jump_pc = prev_pc;

    reg[63:0] next_seq_pc;
    always @(*) begin
        casex(imem_data[63:62])
        2'b0x: next_seq_pc = prev_pc + 2;
        2'b10: next_seq_pc = prev_pc + 4;
        2'b11: next_seq_pc = prev_pc + 8;
        endcase
    end

    always @(*) begin
        pc = prev_pc;
        if(do_jump) pc = jump_pc;
        else if(stall) pc = prev_pc;
        else if(imem_data_valid) begin
            pc = next_seq_pc;
        end
    end

    reg[63:0] next_data;
    always @(*) begin
        next_data = inst_data;
        if(imem_data_valid) next_data = imem_data;
    end

    reg[63:0] prev_data;
    reg just_stalled;
    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            just_stalled <= 0;
            inst_data <= 64'h0;
        end else begin
            just_stalled <= stall;
            if(~imem_data_valid | do_jump) inst_data <= 64'h0;
            else if(~stall) inst_data <= imem_data;
        end
    end
```

(continues on next page)

(continued from previous page)

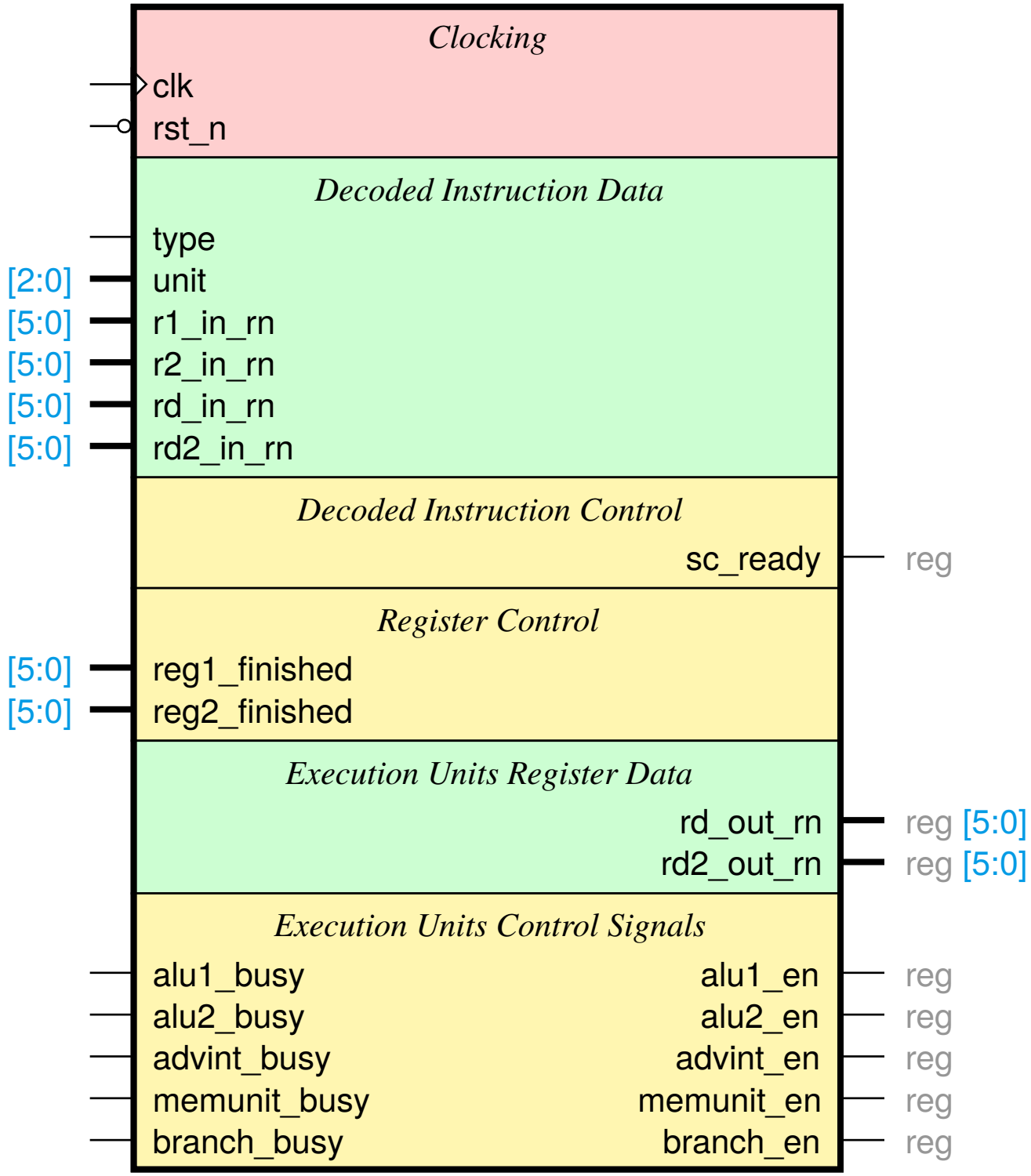
```
82
83      //assign inst_data = ~imem_data_valid ? 64'h0 :  stall|just_stalled ? prev_data :␣
    ↪imem_data;
84      assign imem_addr_valid = 1;
85
86  endmodule
```

5.2.13 ff_sync.v




```
//Copyright 2017 Christopher Parish
//
//Licensed under the Apache License, Version 2.0 (the "License");
//you may not use this file except in compliance with the License.
//You may obtain a copy of the License at
//
//   http://www.apache.org/licenses/LICENSE-2.0
//
//Unless required by applicable law or agreed to in writing, software
//distributed under the License is distributed on an "AS IS" BASIS,
//WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
//See the License for the specific language governing permissions and
//limitations under the License.

module ff_sync #(parameter WIDTH=1)(
    input clk,
    input rst_p,
    input[WIDTH-1:0] in_async,
    output reg[WIDTH-1:0] out);

    (* ASYNC_REG = "TRUE" *) reg[WIDTH-1:0] sync_reg;
    always @(posedge clk, posedge rst_p) begin
        if(rst_p) begin
            sync_reg <= 0;
            out <= 0;
        end else begin
            {out, sync_reg} <= {sync_reg, in_async};
        end
    end

endmodule
```

5.2.14 schedule.v




```
1 //Raisin64 Instruction Scheduler
2 //Schedules a ready instruction to a free execution unit capiable of servicing it
```

(continues on next page)

(continued from previous page)

```
3
4   module schedule(
5       //# {{clocks|Clocking}}
6       input clk,
7       input rst_n,
8
9       //# {{data|Decoded Instruction Data}}
10      input type,
11      input[2:0] unit,
12      input[5:0] r1_in_rn,
13      input[5:0] r2_in_rn,
14      input[5:0] rd_in_rn,
15      input[5:0] rd2_in_rn,
16
17      //# {{control|Decoded Instruction Control}}
18      output reg sc_ready,
19
20      //# {{control|Register Control}}
21      input[5:0] reg1_finished,
22      input[5:0] reg2_finished,
23
24      //# {{data|Execution Units Register Data}}
25      output reg[5:0] rd_out_rn,
26      output reg[5:0] rd2_out_rn,
27
28      //# {{control|Execution Units Control Signals}}
29      output reg alu1_en,
30      output reg alu2_en,
31      output reg advint_en,
32      output reg memunit_en,
33      output reg branch_en,
34
35      input alu1_busy,
36      input alu2_busy,
37      input advint_busy,
38      input memunit_busy,
39      input branch_busy
40      );
41
42      reg[63:0] reg_busy;
43
44      wire alu_type, advint_type, memunit_type, branch_type;
45      assign alu_type = ~unit[2];
46      assign advint_type = ~type && unit==3'h4;
47      assign memunit_type = type && (unit==3'h4 | unit==3'h5 | unit==3'h6);
48      assign branch_type = unit==3'h7;
49
50      wire instIssued;
51      assign instIssued = alu1_en | alu2_en | advint_en | memunit_en | branch_en;
52
53      reg operand_unavailable;
54
55      always @(*)
56      begin
57          operand_unavailable = 0;
58
59          //The register was previously busy
```

(continues on next page)

(continued from previous page)

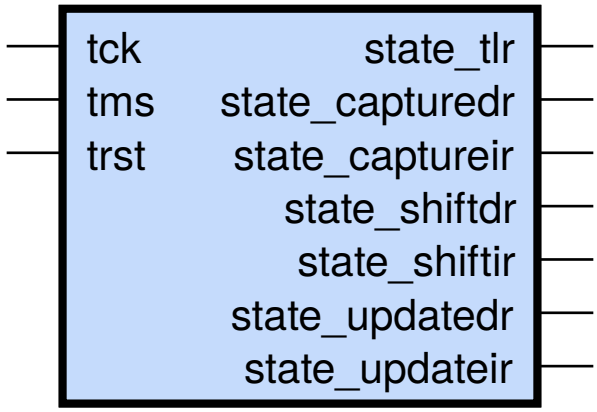
```
60          if(reg_busy[r1_in_rn] && r1_in_rn!=reg1_finished && r1_in_rn!=reg2_finished)␣
    ↪operand_unavailable = 1;
61          else if(reg_busy[r2_in_rn] && r2_in_rn!=reg2_finished && r2_in_rn!=reg1_
    ↪finished) operand_unavailable = 1;
62
63          //We just issued something to an execution unit
64          else if(instIssued) begin
65              //The incoming source register is non-zero
66              if(|r1_in_rn) begin
67                  //And it matches the previous destination register number.  We
68                  //will stall here until it is picked up by reg_busy next cycle
69                  if(rd_out_rn==r1_in_rn) operand_unavailable = 1;
70                  else if(rd2_out_rn==r1_in_rn) operand_unavailable = 1;
71
72              end
73              if(|r2_in_rn) begin
74                  if(rd_out_rn==r2_in_rn) operand_unavailable = 1;
75                  else if(rd2_out_rn==r2_in_rn) operand_unavailable = 1;
76              end
77          end
78      end
79
80      always @(*)
81      begin
82          sc_ready = 0;
83          if(~operand_unavailable & ~branch_busy) begin //TODO Branch busy condition␣
    ↪can probably be moved outside the module via muxing of the unit_en lines with the␣
    ↪operation cancel signal
84              if(alu_type & (~alu1_busy | ~alu2_busy)) sc_ready = 1;
85              else if(advint_type & ~advint_busy) sc_ready = 1;
86              else if(memunit_type & ~memunit_busy) sc_ready = 1;
87              else if(branch_type) sc_ready = 1;
88          end
89      end
90
91      always @(posedge clk or negedge rst_n)
92      begin
93          if(~rst_n) begin
94              alu1_en <= 0;
95              alu2_en <= 0;
96              advint_en <= 0;
97              memunit_en <= 0;
98              branch_en <= 0;
99              rd_out_rn <= 6'h0;
100             rd2_out_rn <= 6'h0;
101             reg_busy <= 64'h0;
102
103         end else begin
104             //Only allow the scheduling of instructions if the source registers
105             //aren't the destination of in-progress instructions.
106             alu1_en <= 0;
107             alu2_en <= 0;
108             advint_en <= 0;
109             memunit_en <= 0;
110             branch_en <= 0;
111
112             reg_busy[reg1_finished] <= 0;
```

(continues on next page)

(continued from previous page)

```
113             reg_busy[reg2_finished] <= 0;
114
115             if(~operand_unavailable & ~branch_busy) begin
116                 if(alu_type & ~alu1_busy) begin
117                     alu1_en <= 1;
118                     rd_out_rn <= rd_in_rn;
119                     if(|rd_in_rn) reg_busy[rd_in_rn] <= 1;
120
121                 end else if(alu_type & ~alu2_busy) begin
122                     alu2_en <= 1;
123                     rd_out_rn <= rd_in_rn;
124                     if(|rd_in_rn) reg_busy[rd_in_rn] <= 1;
125
126                 end else if(advint_type & ~advint_busy) begin
127                     advint_en <= 1;
128                     rd_out_rn <= rd_in_rn;
129                     rd2_out_rn <= rd2_in_rn;
130                     if(|rd_in_rn) reg_busy[rd_in_rn] <= 1;
131                     if(|rd2_in_rn) reg_busy[rd2_in_rn] <= 1;
132
133                 end else if(memunit_type & ~memunit_busy) begin
134                     memunit_en <= 1;
135                     rd_out_rn <= rd_in_rn;
136                     if(|rd_in_rn && unit!=3'h6) reg_busy[rd_in_rn] <= 1;
137
138                 end else if(branch_type) begin
139                     branch_en <= 1;
140                     rd_out_rn <= rd_in_rn;
141                     //No need to mark R63 busy. If the branch is taken, other
142                     //instructions in the pipeline are cancelled.
143                 end
144             end
145         end
146     end
147
148 endmodule
```

5.2.15 jtag_state_machine.v



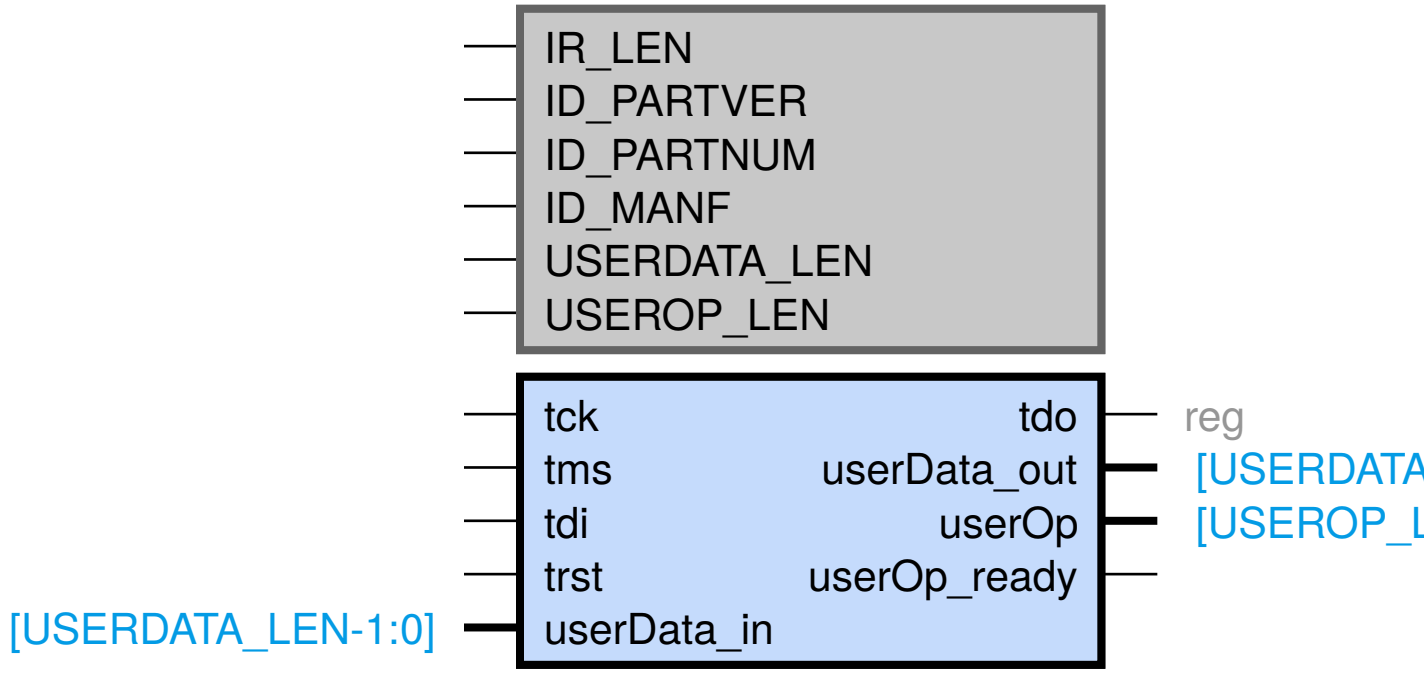

```
/* jtag_state_machine.v
 *
 * JTAG TAP State Machine
 *
 *------------------------------------------------------------------------------
 *
 * Copyright 2018 Christopher Parish
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

module jtag_state_machine(
        input tck,
        input tms,
        input trst,

        output state_tlr,
        output state_capturedr,
        output state_captureir,
        output state_shiftdr,
        output state_shiftir,
        output state_updatedr,
        output state_updateir

    );

    localparam TEST_LOGIC_RESET = 4'h0;
    localparam RUN_TEST_IDLE    = 4'h1;
    localparam SELECT_DR        = 4'h2;
    localparam CAPTURE_DR       = 4'h3;
```

(continues on next page)

(continued from previous page)

```
    localparam SHIFT_DR         = 4'h4;
    localparam EXIT1_DR         = 4'h5;
    localparam PAUSE_DR         = 4'h6;
    localparam EXIT2_DR         = 4'h7;
    localparam UPDATE_DR        = 4'h8;
    localparam SELECT_IR        = 4'h9;
    localparam CAPTURE_IR       = 4'hA;
    localparam SHIFT_IR         = 4'hB;
    localparam EXIT1_IR         = 4'hC;
    localparam PAUSE_IR         = 4'hD;
    localparam EXIT2_IR         = 4'hE;
    localparam UPDATE_IR        = 4'hF;

    reg[3:0] state;

    always @(posedge tck or negedge trst) begin
        if(~trst) begin
            state <= TEST_LOGIC_RESET;
        end else begin
            case(state)
                TEST_LOGIC_RESET: state <= tms ? TEST_LOGIC_RESET : RUN_TEST_IDLE;
                RUN_TEST_IDLE:    state <= tms ? SELECT_DR : RUN_TEST_IDLE;
                SELECT_DR :       state <= tms ? SELECT_IR : CAPTURE_DR;
                CAPTURE_DR :      state <= tms ? EXIT1_DR : SHIFT_DR;
                SHIFT_DR:         state <= tms ? EXIT1_DR : SHIFT_DR;
                EXIT1_DR:         state <= tms ? UPDATE_DR : PAUSE_DR;
                PAUSE_DR:         state <= tms ? EXIT2_DR : PAUSE_DR;
                EXIT2_DR:         state <= tms ? UPDATE_DR : SHIFT_DR;
                UPDATE_DR:        state <= tms ? SELECT_DR : RUN_TEST_IDLE;
                SELECT_IR:        state <= tms ? TEST_LOGIC_RESET : CAPTURE_IR;
                CAPTURE_IR:       state <= tms ? EXIT1_IR : SHIFT_IR;
                SHIFT_IR :        state <= tms ? EXIT1_IR : SHIFT_IR;
                EXIT1_IR:         state <= tms ? UPDATE_IR : PAUSE_IR;
                PAUSE_IR:         state <= tms ? EXIT2_IR : PAUSE_IR;
                EXIT2_IR:         state <= tms ? UPDATE_IR : SHIFT_IR;
                UPDATE_IR:        state <= tms ? SELECT_DR : RUN_TEST_IDLE;
            endcase
        end
    end

    //I was going to use a function, but Vivado pooped itself when I tried. Typical...
    assign state_tlr = (state == TEST_LOGIC_RESET);
    assign state_capturedr = (state == CAPTURE_DR);
    assign state_captureir = (state == CAPTURE_IR);
    assign state_shiftdr = (state == SHIFT_DR);
    assign state_shiftir = (state == SHIFT_IR);
    assign state_updatedr = (state == UPDATE_DR);
    assign state_updateir = (state == UPDATE_IR);

endmodule
```

5.2.16 jtaglet.v




```
/* jtaglet.v
 *
 * Top module for the JTAGlet JTAG TAP project
 *
 *--------------------------------------------------------------------------------
 *
 * Copyright 2018 Christopher Parish
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

module jtaglet #(
    parameter IR_LEN = 4,
    parameter ID_PARTVER = 4'h0,
    parameter ID_PARTNUM = 16'h0000,
    parameter ID_MANF = 11'h000,
    parameter USERDATA_LEN = 32,
    parameter USEROP_LEN = 8
    )(
    input tck,
    input tms,
```

(continues on next page)

(continued from previous page)

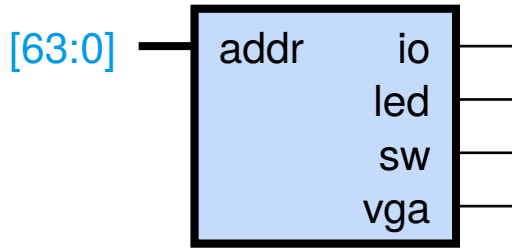
```
    input tdi,
    output reg tdo,
    input trst,

    input[USERDATA_LEN-1:0] userData_in,
    output[USERDATA_LEN-1:0] userData_out,
    output[USEROP_LEN-1:0] userOp,
    output userOp_ready
    );

    localparam USERDATA_OP = 4'b1000;
    localparam USEROP_OP = 4'b1001;
    localparam IDCODE_OP = {{(IR_LEN-1){1'b1}},1'b0}; //e.g. b1110
    localparam BYPASS_OP = {IR_LEN{1'b1}};// e.g. b1111 (required bit pattern per spec)

    wire[31:0] idcode = {ID_PARTVER, ID_PARTNUM, ID_MANF, 1'b1};

    wire state_tlr, state_capturedr, state_captureir, state_shiftdr, state_shiftir,
        state_updatedr, state_updateir;

    jtag_state_machine jsm(.tck(tck), .tms(tms), .trst(trst), .state_tlr(state_tlr),
        .state_capturedr(state_capturedr), .state_captureir(state_captureir),
        .state_shiftdr(state_shiftdr), .state_shiftir(state_shiftir),
        .state_updatedr(state_updatedr), .state_updateir(state_updateir));

    reg[IR_LEN-1:0] ir_reg;

    //USERDATA - DR becomes a USERDATA_LEN bit user data register passed out of the module
    wire userData_tdo;
    jtag_reg #(.IR_LEN(IR_LEN), .DR_LEN(USERDATA_LEN), .IR_OPCODE(USERDATA_OP)) userData_reg
        (.tck(tck), .trst(trst), .tdi(tdi), .tdo(userData_tdo), .state_tlr(state_tlr),
         .state_capturedr(state_capturedr), .state_shiftdr(state_shiftdr),
         .state_updatedr(state_updatedr), .ir_reg(ir_reg), .dr_dataOut(userData_out),
         .dr_dataIn(userData_in), .dr_dataOutReady());

    //USEROPCODE - DR becomes an 8 bit operation select/initiate register passed out of the module
    wire userOp_tdo;
    jtag_reg #(.IR_LEN(IR_LEN), .DR_LEN(USEROP_LEN), .IR_OPCODE(USEROP_OP)) userOp_reg
        (.tck(tck), .trst(trst), .tdi(tdi), .tdo(userOp_tdo), .state_tlr(state_tlr),
         .state_capturedr(state_capturedr), .state_shiftdr(state_shiftdr),
         .state_updatedr(state_updatedr), .ir_reg(ir_reg), .dr_dataOut(userOp),
         .dr_dataIn(8'b0), .dr_dataOutReady(userOp_ready));

    //IDCODE - DR is pre-loaded with the 32 bit identification code of this part
    wire idcode_tdo;
    jtag_reg #(.IR_LEN(IR_LEN), .DR_LEN(32), .IR_OPCODE(IDCODE_OP)) idcode_reg
        (.tck(tck), .trst(trst), .tdi(tdi), .tdo(idcode_tdo), .state_tlr(state_tlr),
         .state_capturedr(state_capturedr), .state_shiftdr(state_shiftdr),
         .state_updatedr(1'b0), .ir_reg(ir_reg), .dr_dataOut(),
         .dr_dataIn(idcode), .dr_dataOutReady());

    //BYPASS - DR becomes a 1 bit wide register, suitable for bypassing this part
    wire bypass_tdo;
```

(continues on next page)

(continued from previous page)

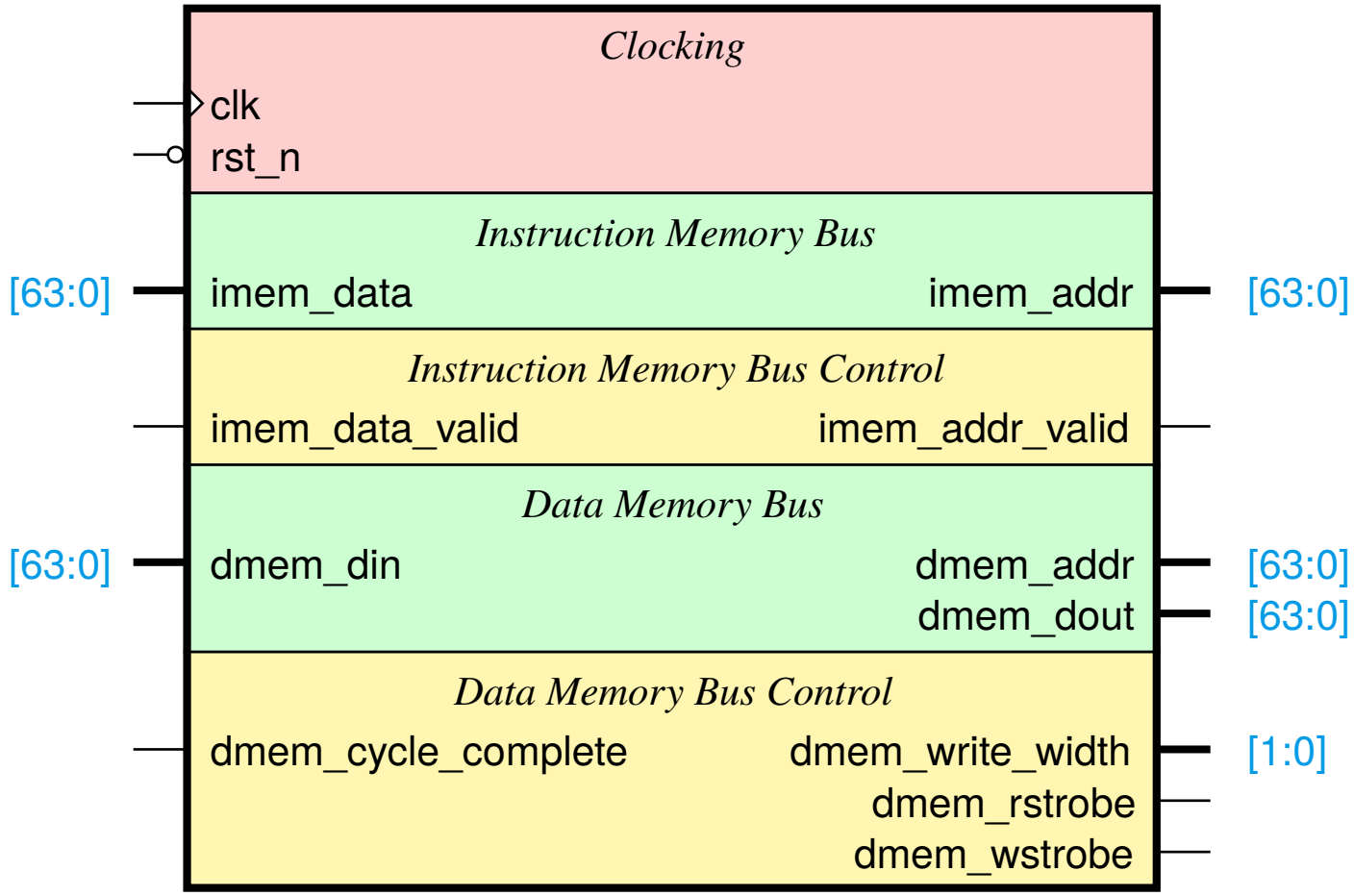
```
85      jtag_reg #(.IR_LEN(IR_LEN), .DR_LEN(1), .IR_OPCODE(BYPASS_OP)) bypass_reg
86           (.tck(tck), .trst(trst), .tdi(tdi), .tdo(bypass_tdo), .state_tlr(state_tlr),
87            .state_capturedr(state_capturedr), .state_shiftdr(state_shiftdr),
88            .state_updatedr(1'b0), .ir_reg(ir_reg), .dr_dataOut(),
89            .dr_dataIn(1'b0), .dr_dataOutReady());
90
91      //Instruction Register
92      wire ir_tdo;
93      assign ir_tdo = ir_reg[0];
94      always @(posedge tck or negedge trst) begin
95          if(~trst) begin
96              ir_reg <= IDCODE_OP;
97          end else if(state_tlr) begin
98              ir_reg <= IDCODE_OP;
99          end else if(state_captureir) begin
100             //We need to load the BYPASS reg with seq ending in 01.
101             ir_reg <= {{(IR_LEN-1){1'b0}},1'b1}; //e.g. b0001
102         end else if(state_shiftir) begin
103             ir_reg <= {tdi, ir_reg[IR_LEN-1:1]};
104         end
105     end
106
107     //IR selects the appropriate DR
108     reg tdo_pre;
109     always @(*) begin
110         tdo_pre = 0;
111         if(state_shiftdr) begin
112             case(ir_reg)
113                 IDCODE_OP:      tdo_pre = idcode_tdo;
114                 BYPASS_OP:      tdo_pre = bypass_tdo;
115                 USERDATA_OP:    tdo_pre = userData_tdo;
116                 USEROP_OP:      tdo_pre = userOp_tdo;
117                 default:        tdo_pre = bypass_tdo;
118             endcase
119         end else if(state_shiftir) begin
120             tdo_pre = ir_tdo;
121         end
122     end
123
124     //TDO updates on the negative edge according to the spec
125     always @(negedge tck)
126     begin
127         tdo <= tdo_pre;
128     end
129
130 endmodule
```

5.2.17 memory_map.v




```
//Raisin64 - Memory Map
module memory_map(
    input[63:0] addr,
    output io,
    output led,
    output sw,
    output vga
    );

    //As physical addresses are sign-extended from the 47th bit,
    //only bits 46:14 really matter

    //Upper-half of the memory map is IO for now
    assign io = addr[46];

    assign led = (addr[46:14] == 33'h100000001);
    assign sw  = (addr[46:14] == 33'h100000002);
    assign vga = (addr[46:18] == 33'h10000001);

endmodule
```

5.2.18 pipeline.v




```
1  //Raisin64 - Pipeline
2
3  module pipeline(
4      //# {{clocks|Clocking}}
5      input clk,
6      input rst_n,
7
8      //# {{data|Instruction Memory Bus}}
9      output[63:0] imem_addr,
10     input[63:0] imem_data,
11
12     //# {{control|Instruction Memory Bus Control}}
13     input imem_data_valid,
14     output imem_addr_valid,
15
16     //# {{data|Data Memory Bus}}
17     output[63:0] dmem_addr,
18     output[63:0] dmem_dout,
19     input[63:0] dmem_din,
20
21     //# {{control|Data Memory Bus Control}}
22     input dmem_cycle_complete,
```

(continues on next page)

(continued from previous page)

```
    output[1:0] dmem_write_width,
    output dmem_rstrobe,
    output dmem_wstrobe
    );

    wire sc_ready;
    wire[63:0] jump_pc;
    wire do_jump;
    wire cancel_now;
    assign cancel_now = do_jump;

    /////////  FETCH    /////////
    wire[63:0] fe_inst;
    wire[63:0] fe_next_pc;

    fetch fetch1(
        .clk(clk),
        .rst_n(rst_n),
        .imem_addr(imem_addr),
        .imem_data(imem_data),
        .imem_data_valid(imem_data_valid),
        .imem_addr_valid(imem_addr_valid),
        .inst_data(fe_inst),
        .next_jump_pc(fe_next_pc),
        .jump_pc(jump_pc), .do_jump(do_jump),
        .stall(~sc_ready)
        );

    reg fe_cancelled;
    always @(posedge clk or negedge rst_n) begin
        if(~rst_n) fe_cancelled <= 0;
        else fe_cancelled <= cancel_now;
    end

    /////////  DECODE   /////////

    wire de_type;
    wire[2:0] de_unit;
    wire[1:0] de_op;
    wire[5:0] de_rd_rn;
    wire[5:0] de_rd2_rn;
    wire[5:0] de_rs1_rn;
    wire[5:0] de_rs2_rn;
    wire[63:0] de_imm_data;

    wire[5:0] de_r1_rn;
    wire[5:0] de_r2_rn;

    decode decode1(
        .clk(clk), .rst_n(rst_n),
        .instIn(fe_inst),
        .type(de_type),
        .unit(de_unit),
        .op(de_op),
        .rs1_rn(de_rs1_rn), .rs2_rn(de_rs2_rn),
        .rd_rn(de_rd_rn), .rd2_rn(de_rd2_rn),
        .imm_data(de_imm_data),
```

(continues on next page)

(continued from previous page)

```
        .r1_rn(de_r1_rn), .r2_rn(de_r2_rn),
        .stall(~sc_ready), .cancel(cancel_now|fe_cancelled)
        );

    //Delay the Next PC to the schedule stage
    reg[63:0] de_next_pc;
    reg de_cancelled;

    always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) begin
            de_next_pc <= 64'h0;
            de_cancelled <= 0;
        end else begin
            de_cancelled <= cancel_now|fe_cancelled;
            if (sc_ready) begin
                if(cancel_now|fe_cancelled) de_next_pc <= 64'h0;
                else de_next_pc <= fe_next_pc;
            end
        end
    end

    ////////// REG FILE  //////////
    //Concurrent with Schedule phase

    wire[63:0] rf_writeback;
    wire[63:0] rf_data1;
    wire[63:0] rf_data2;
    wire[5:0] rf_writeback_rn;

    //Register file selected by the scheduler, registered in the execute stage
    //and written to during commit.
    regfile regfile1(
        .clk(clk), .rst_n(rst_n), .w_data(rf_writeback), .r1_data(rf_data1),
        .r2_data(rf_data2), .r1_rn(de_r1_rn), .r2_rn(de_r2_rn),
        .w_rn(rf_writeback_rn), .w_en(|rf_writeback_rn)
        );

    ////////// SCHEDULE  //////////
    //Concurrent with Register File

    wire sc_alu1_en;
    wire sc_alu2_en;
    wire sc_advint_en;
    wire sc_memunit_en;
    wire sc_branch_en;

    wire sc_alu1_busy;
    wire sc_alu2_busy;
    wire sc_advint_busy;
    wire sc_memunit_busy;
    wire sc_branch_busy;

    wire[5:0] sc_rd_rn;
    wire[5:0] sc_rd2_rn;

    schedule schedule1(
```

(continues on next page)

(continued from previous page)

```
137         .clk(clk), .rst_n(rst_n),
138         .type(de_type), .unit(cancel_now|de_cancelled ? 3'h0 : de_unit),
139         .r1_in_rn(de_r1_rn), .r2_in_rn(de_r2_rn),
140         .rd_in_rn(cancel_now|de_cancelled ? 6'h0 : de_rd_rn),
141         .rd2_in_rn(cancel_now|de_cancelled ? 6'h0 : de_rd2_rn),
142         .sc_ready(sc_ready),
143         .rd_out_rn(sc_rd_rn), .rd2_out_rn(sc_rd2_rn),
144
145         .reg1_finished(rf_writeback_rn), .reg2_finished(6'h0),
146
147         .alu1_en(sc_alu1_en), .alu2_en(sc_alu2_en), .advint_en(sc_advint_en),
148         .memunit_en(sc_memunit_en), .branch_en(sc_branch_en),
149
150         .alu1_busy(sc_alu1_busy), .alu2_busy(sc_alu2_busy),
151         .advint_busy(sc_advint_busy), .memunit_busy(sc_memunit_busy),
152         .branch_busy(sc_branch_busy)
153         );
154
155     //Delay the relevant decode data used by execution through the schedule phase
156     reg sc_type;
157     reg[2:0] sc_unit;
158     reg[1:0] sc_op;
159     reg[63:0] sc_imm_data;
160     reg[63:0] sc_next_pc;
161
162     always @(posedge clk or negedge rst_n)
163     begin
164         if(~rst_n) begin
165             sc_type <= 0;
166             sc_unit <= 3'h0;
167             sc_op <= 2'h0;
168             sc_imm_data <= 64'h0;
169             sc_next_pc <= 64'h0;
170         end else begin
171             if(cancel_now|de_cancelled) sc_unit <= 3'h0;
172             else sc_unit <= de_unit;
173
174             sc_type <= de_type;
175             sc_op <= de_op;
176             sc_imm_data <= de_imm_data;
177             sc_next_pc <= de_next_pc;
178         end
179     end
180
181     //////////  EXECUTE  //////////
182
183     wire[63:0] ex_alu1_result;
184     wire[63:0] ex_alu2_result;
185     wire[63:0] ex_advint_result;
186     wire[63:0] ex_advint_result2;
187     wire[63:0] ex_memunit_result;
188     wire[63:0] ex_branch_r63;
189
190     wire[5:0] ex_alu1_rd_rn;
191     wire[5:0] ex_alu2_rd_rn;
192     wire[5:0] ex_advint_rd_rn;
193     wire[5:0] ex_advint_rd2_rn;
```

(continues on next page)

(continued from previous page)

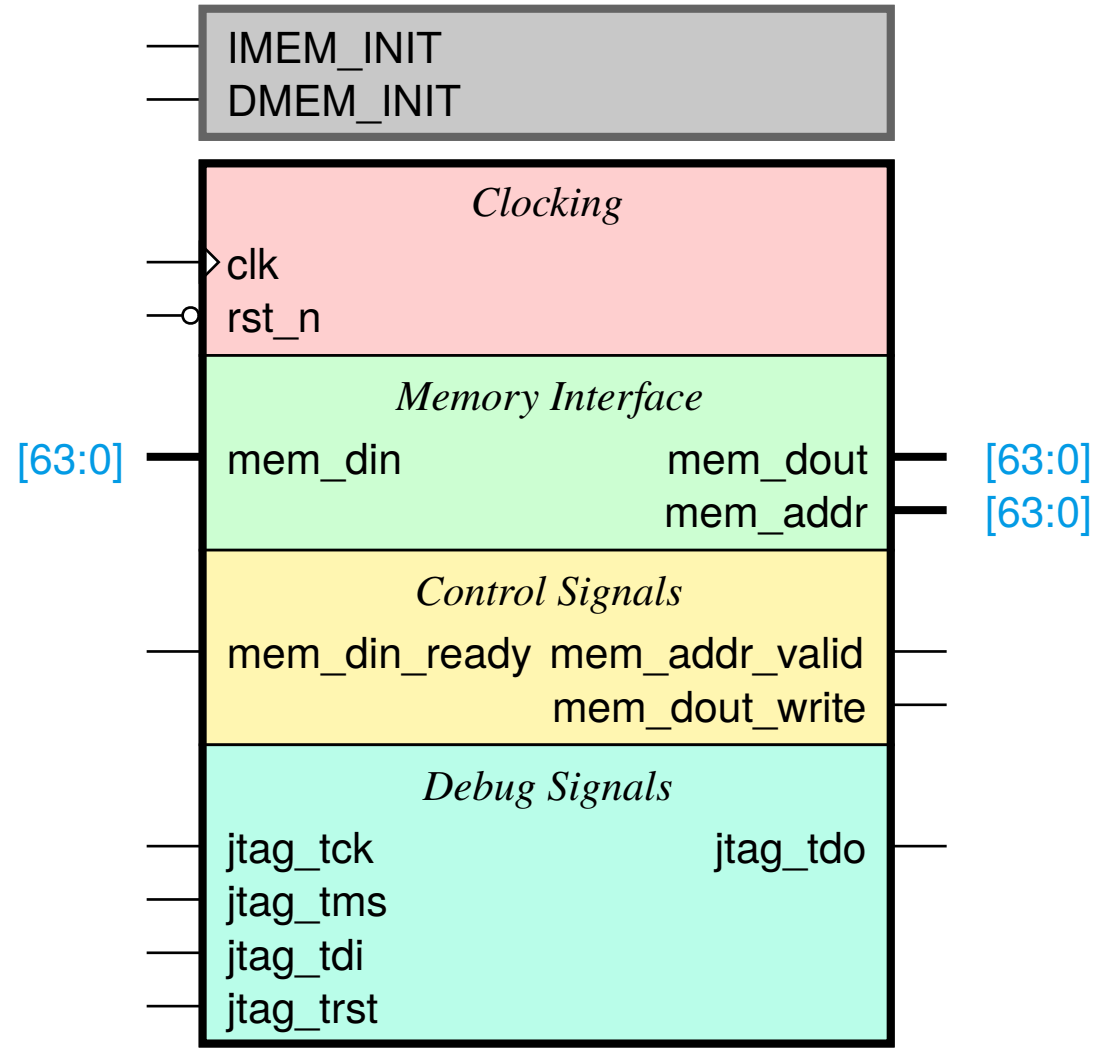
```
194     wire[5:0] ex_memunit_rd_rn;
195     wire ex_branch_r63_update;
196 
197     wire ex_alu1_valid;
198     wire ex_alu2_valid;
199     wire ex_advint_valid;
200     wire ex_memunit_valid;
201 
202     wire ex_alu1_stall;
203     wire ex_alu2_stall;
204     wire ex_advint_stall;
205     wire ex_memunit_stall;
206     wire ex_branch_stall;
207 
208     ex_alu ex_alu1(
209         .clk(clk), .rst_n(rst_n), .in1(rf_data1), .in2(sc_type ? sc_imm_data : rf_
    ↪data2), .out(ex_alu1_result),
210         .ex_enable(sc_alu1_en), .ex_busy(sc_alu1_busy), .rd_in_rn(sc_rd_rn), .unit(sc_
    ↪unit),
211         .op(sc_op), .rd_out_rn(ex_alu1_rd_rn), .valid(ex_alu1_valid), .stall(ex_alu1_
    ↪stall)
212         );
213 
214     ex_alu ex_alu2(
215         .clk(clk), .rst_n(rst_n), .in1(rf_data1), .in2(sc_type ? sc_imm_data : rf_
    ↪data2), .out(ex_alu2_result),
216         .ex_enable(sc_alu2_en), .ex_busy(sc_alu2_busy), .rd_in_rn(sc_rd_rn), .unit(sc_
    ↪unit),
217         .op(sc_op), .rd_out_rn(ex_alu2_rd_rn), .valid(ex_alu2_valid), .stall(ex_alu2_
    ↪stall)
218         );
219 
220     ex_advint ex_advint(
221         .clk(clk), .rst_n(rst_n),
222         .in1(rf_data1), .in2(rf_data2),
223         .out(ex_advint_result), .out2(ex_advint_result2),
224         .ex_enable(sc_advint_en), .ex_busy(sc_advint_busy),
225         .rd_in_rn(sc_rd_rn), .rd2_in_rn(sc_rd2_rn),
226         .unit(sc_unit), .op(sc_op),
227         .rd_out_rn(ex_advint_rd_rn), .rd2_out_rn(ex_advint_rd2_rn),
228         .valid(ex_advint_valid), .stall(ex_advint_stall)
229         );
230 
231     ex_memory ex_memory1(
232         .clk(clk), .rst_n(rst_n),
233         .dmem_din(dmem_din), .dmem_dout(dmem_dout), .dmem_addr(dmem_addr),
234         .dmem_cycle_complete(dmem_cycle_complete),
235         .dmem_width(dmem_write_width),
236         .dmem_rstrobe(dmem_rstrobe), .dmem_wstrobe(dmem_wstrobe),
237         .base(rf_data1), .data(rf_data2), .offset(sc_imm_data[31:0]),
238         .out(ex_memunit_result),
239         .ex_enable(sc_memunit_en), .ex_busy(sc_memunit_busy),
240         .rd_in_rn(sc_rd_rn), .unit(sc_unit), .op(sc_op),
241         .rd_out_rn(ex_memunit_rd_rn), .valid(ex_memunit_valid),
242         .stall(ex_memunit_stall)
243         );
244 
```

(continues on next page)

(continued from previous page)

```
245     ex_branch ex_branch1(
246         .clk(clk), .rst_n(rst_n),
247         .in1(rf_data1), .in2(rf_data2), .imm(sc_imm_data),
248         .next_pc(sc_next_pc), .jump_pc(jump_pc), .do_jump(do_jump),
249         .r63(ex_branch_r63), .r63_update(ex_branch_r63_update),
250         .ex_enable(sc_branch_en), .ex_busy(sc_branch_busy),
251         .stall(ex_branch_stall),
252         .op(sc_op), .type(sc_type)
253         );
254 
255     //////////  COMMIT   //////////
256 
257     commit commit1(
258         .clk(clk), .rst_n(rst_n),
259 
260         .alu1_result(ex_alu1_result),
261         .alu2_result(ex_alu2_result),
262         .advint_result(ex_advint_result),
263         .advint_result2(ex_advint_result2),
264         .memunit_result(ex_memunit_result),
265         .branch_result(ex_branch_r63),
266 
267         .alu1_rn(ex_alu1_rd_rn),
268         .alu2_rn(ex_alu2_rd_rn),
269         .advint_rn(ex_advint_rd_rn),
270         .advint_rn2(ex_advint_rd2_rn),
271         .memunit_rn(ex_memunit_rd_rn),
272 
273         .alu1_valid(ex_alu1_valid),
274         .alu2_valid(ex_alu2_valid),
275         .advint_valid(ex_advint_valid),
276         .memunit_valid(ex_memunit_valid),
277         .branch_valid(ex_branch_r63_update),
278 
279         .alu1_stall(ex_alu1_stall),
280         .alu2_stall(ex_alu2_stall),
281         .advint_stall(ex_advint_stall),
282         .memunit_stall(ex_memunit_stall),
283         .branch_stall(ex_branch_stall),
284 
285         .write_data(rf_writeback), .write_rn(rf_writeback_rn)
286         );
287 
288 endmodule
```

5.2.19 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,
```

(continues on next page)

(continued from previous page)

```
    //# {{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;

    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)
        );

```

(continues on next page)

(continued from previous page)

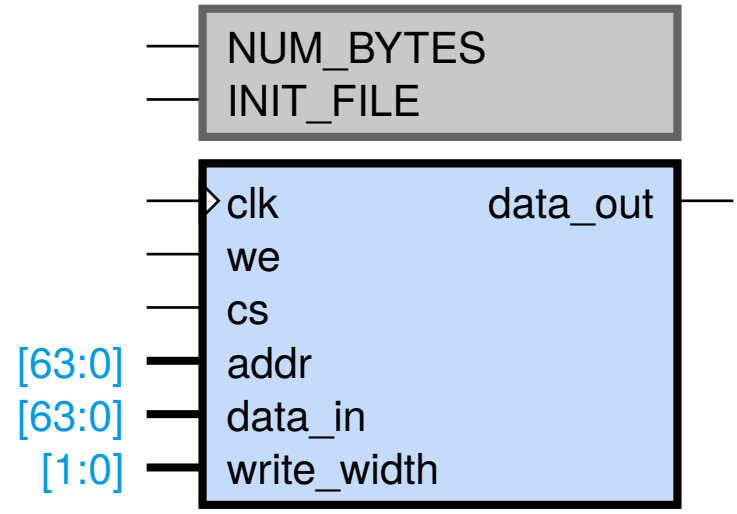
```
76
77      //////////  Data RAM  //////////
78      wire[63:0] effective_dmem_addr;
79      wire[63:0] effective_dmem_to_ram;
80
81      wire[63:0] dmem_addr;
82      wire[63:0] dmem_to_ram;
83      wire[63:0] dmem_to_cpu;
84      wire[63:0] dmem_from_ram;
85      wire[1:0] dmem_write_width;
86      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, there
104     //is no reason to create special logic to handle misaligned accesses into data space
105     //in case an IO unit requires it (the ram modules handle this condition internally).
106     //Instead we simply state misaligned IO access it is unsupported (for now).
107     always @(*) begin
108         if((dmem_rstrobe|dmem_wstrobe) & io_space & |dmem_write_width & ~clk) begin
109             $display("Unaligned data IO access not supported in this revision");
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),
```

(continues on next page)

(continued from previous page)

```
        .INIT_FILE(DMEM_INIT)
        ) dmem (
        .clk(clk),
        .we(~io_space & (dmem_wstrobe|dbg_dmem_we)), .cs(~io_space & (dmem_
↪wstrobe|dmem_rstrobe|dbg_dmem_ce)),
        .write_width(dmem_write_width),
        .addr(effective_dmem_addr),
        .data_in(effective_dmem_to_ram),
        .data_out(dmem_from_ram)
        );


    //////////  Raisin64 Execution Core  //////////
    pipeline pipeline1(
        .clk(clk),
        .rst_n(cpu_rst_n),
        .imem_addr(imem_addr),
        .imem_data(effective_imem_data_to_cpu),
        .imem_data_valid(imem_data_ready),
        .imem_addr_valid(imem_addr_valid),
        .dmem_addr(dmem_addr), .dmem_dout(dmem_to_ram),
        .dmem_din(dmem_to_cpu),
        .dmem_cycle_complete(dmem_cycle_complete & ~dmem_rstrobe & ~dmem_wstrobe),
        .dmem_write_width(dmem_write_width),
        .dmem_rstrobe(dmem_rstrobe),
        .dmem_wstrobe(dmem_wstrobe)
        );


    //////////  JTAG Module  //////////
    debug_control debug_if(
        .jtag_tck(jtag_tck),
        .jtag_tms(jtag_tms),
        .jtag_tdo(jtag_tdo),
        .jtag_tdi(jtag_tdi),
        .jtag_trst(jtag_trst),
        .cpu_clk(clk),
        .sys_rstn(rst_n),
        .cpu_imem_addr(dbg_imem_addr),
        .cpu_debug_to_imem_data(dbg_imem_to_ram),
        .cpu_imem_to_debug_data(imem_data),
        .cpu_imem_we(dbg_imem_we),
        .cpu_imem_ce(dbg_imem_ce),
        .cpu_dmem_addr(dbg_dmem_addr),
        .cpu_debug_to_dmem_data(dbg_dmem_to_ram),
        .cpu_imem_to_debug_data_ready(dbg_imem_ce & ~dbg_imem_we),
        .cpu_dmem_to_debug_data_ready(dbg_dmem_ce & ~dbg_dmem_we),
        .cpu_dmem_to_debug_data(dmem_to_cpu),
        .cpu_dmem_we(dbg_dmem_we),
        .cpu_dmem_ce(dbg_dmem_ce),
        .cpu_resetn_cpu(dbg_resetn_cpu),
        .cpu_halt_cpu(dbg_halt_cpu)
        );

endmodule
```

5.2.20 ram.v



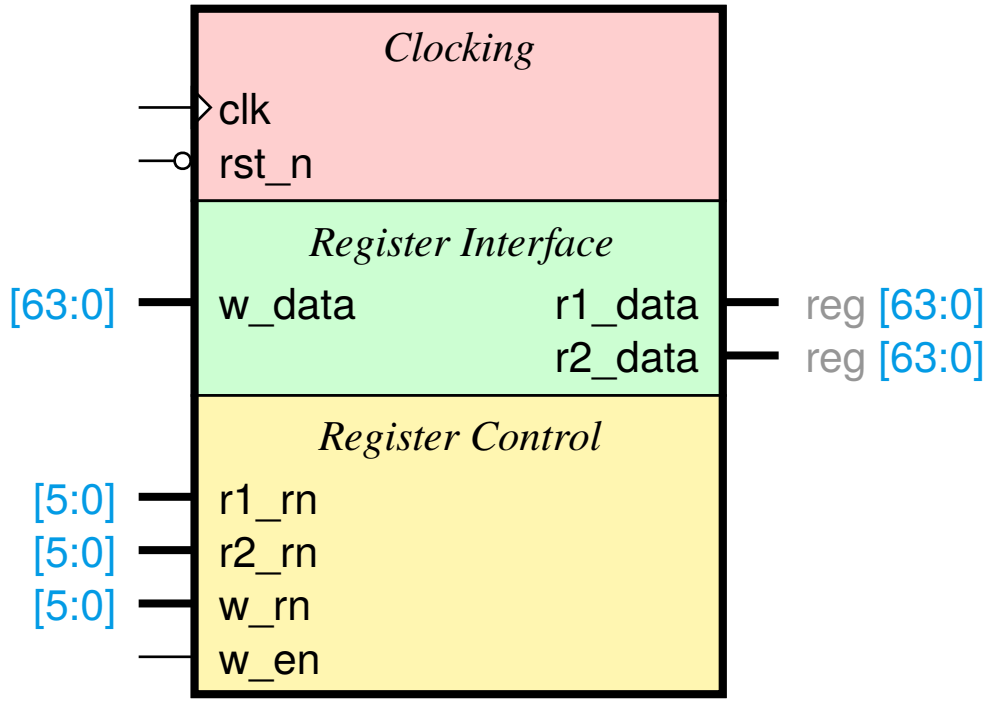

```
1  //Test RAM
2
3  `include "io_def.vh"
4
5  module ram(input clk,
6             input we,
7             input cs,
8             input[63:0] addr,
9             input[63:0] data_in,
10            input[1:0] write_width, //0==64bit, 1==32bit, 2==16bit, 3==8bit
11            output reg[63:0] data_out);
12
13     parameter NUM_BYTES = 0;
14     parameter INIT_FILE = "";
15
16     reg[63:0] ram[0:(NUM_BYTES/8)-1]; //Nx64 RAM
17
18     reg[63:0] ramA_result, ramB_result;
19
20     reg[7:0] weA_b; //Which byte lines are write-enabled
21     reg[7:0] weB_b;
22
23     always @(*) begin
24         weA_b = 8'h00;
25         weB_b = 8'h00;
26         if(we) begin
27             case(write_width)
28             `RAM_WIDTH64: {weA_b,weB_b} = 16'hFF00 >> addr[2:0];
29             `RAM_WIDTH32: {weA_b,weB_b} = 16'hF000 >> addr[2:0];
30             `RAM_WIDTH16: {weA_b,weB_b} = 16'hC000 >> addr[2:0];
31             `RAM_WIDTH8: {weA_b,weB_b} = 16'h8000 >> addr[2:0];
32             endcase
33         end
34     end
35
36     wire[127:0] data_in_shift;
```

(continues on next page)

(continued from previous page)

```
    assign data_in_shift = {data_in, 64'h0} >> addr[2:0]*8;

    generate
    genvar i;
    for(i = 0; i < 8; i = i+1) begin
        always @(posedge clk)
            if(cs)
                if(weA_b[i]) begin
                    ram[addr[63:3]][(i+1)*8-1:i*8] <= data_in_shift[64+(i+1)*8-
↪1:64+i*8];
                end
        end

    for(i = 0; i < 8; i = i+1) begin
        always @(posedge clk)
            if(cs)
                if(weB_b[i]) begin
                    ram[addr[63:3]+1][(i+1)*8-1:i*8] <= data_in_shift[(i+1)*8-1:i*8];
                end
        end
    endgenerate

    //Registered read (ready by next clock cycle)
    always @(posedge clk) begin
        if(cs) begin
            ramA_result <= ram[addr[63:3]];
            ramB_result <= ram[addr[63:3]+1];
        end
    end

    reg[2:0] addr_lsb;
    always @(posedge clk) begin
        addr_lsb <= addr[2:0];
    end

    always @(*) begin
        data_out = ({ramA_result,ramB_result} >> ((8-addr_lsb)*8)) & 64
↪'hFFFFFFFFFFFFFFFF;
    end

    //Populate our program memory with the user-provided hex file
    initial begin
        $readmemh(INIT_FILE, ram);
    end


endmodule
```

5.2.21 regfile.v



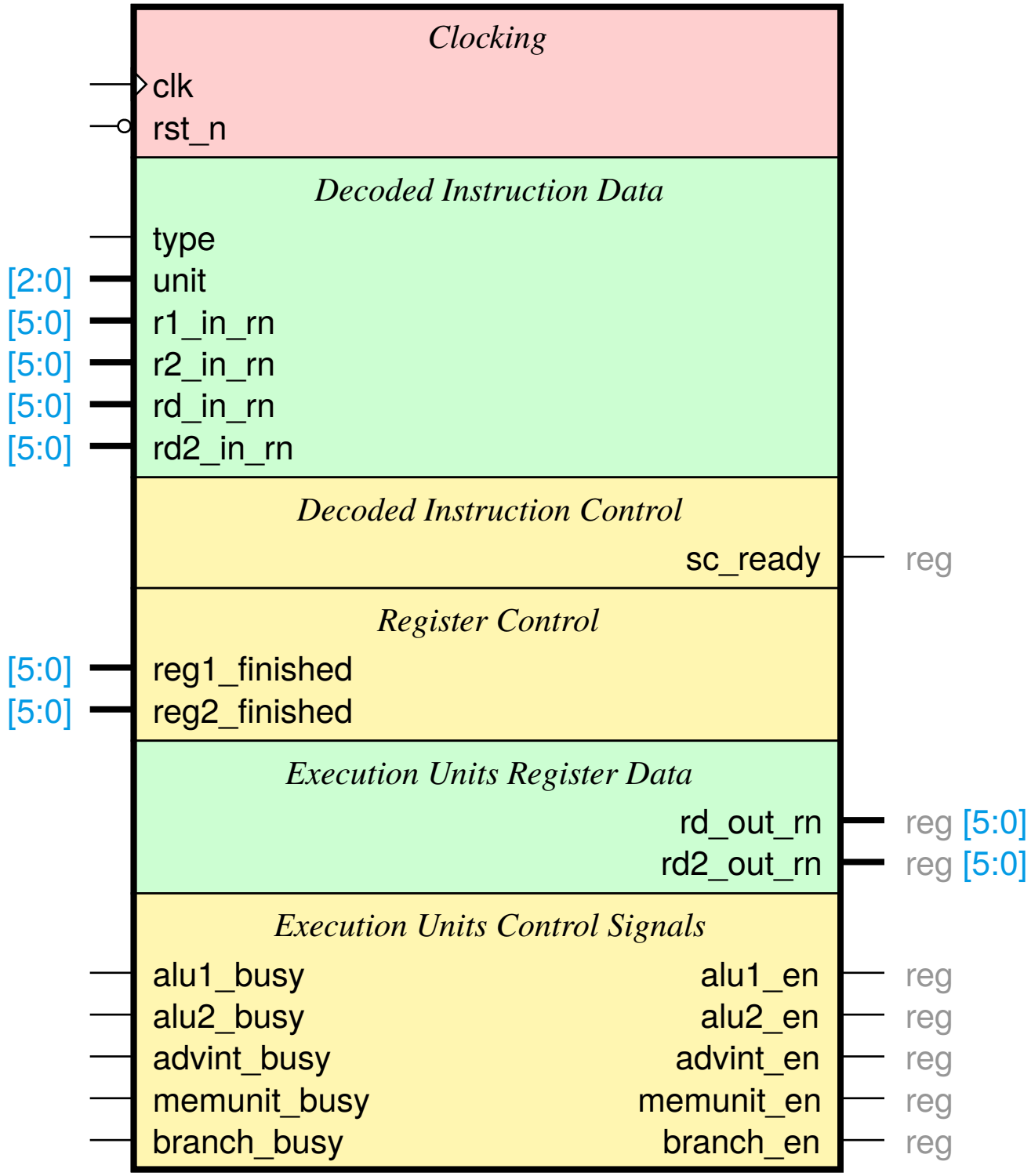

```
1  //Raisin64 Register File
2  //Registered dual-ported register file two read, one write port
3
4  module regfile(
5      //# {{clocks|Clocking}}
6      input clk,
7      input rst_n,
8
9      //# {{data|Register Interface}}
10     input[63:0] w_data,
11     output reg[63:0] r1_data,
12     output reg[63:0] r2_data,
13
14     //# {{control|Register Control}}
15     input[5:0] r1_rn,
16     input[5:0] r2_rn,
17     input[5:0] w_rn,
18     input w_en
19
20     //# {{debug|Debug Signals}}
21     );
22
23     reg[63:0] file[1:63];
24     reg[63:0] r1_data_pre, r2_data_pre;
25     integer i;
26
27     initial for(i = 1; i<64; i = i+1) file[i] <= 64'h0;
28
29     always @(posedge clk) if(w_en && w_rn!=6'h0) file[w_rn] <= w_data;
30
```

(continues on next page)

(continued from previous page)

```
31      always @(posedge clk or negedge rst_n)
32      begin
33          if(~rst_n) r1_data <= 64'h0;
34          else if(r1_rn==6'h0) r1_data <= 64'h0;
35          else r1_data <= (w_rn==r1_rn && w_en) ? w_data : file[r1_rn];
36      end
37
38      always @(posedge clk or negedge rst_n)
39      begin
40          if(~rst_n) r2_data <= 64'h0;
41          else if(r2_rn==6'h0) r2_data <= 64'h0;
42          else r2_data <= (w_rn==r2_rn && w_en) ? w_data : file[r2_rn];
43      end
44  endmodule
```

5.2.22 schedule.v




```
1 //Raisin64 Instruction Scheduler
2 //Schedules a ready instruction to a free execution unit capiable of servicing it
```

(continues on next page)

(continued from previous page)

```

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

    //# {{data|Decoded Instruction Data}}
    input type,
    input[2:0] unit,
    input[5:0] r1_in_rn,
    input[5:0] r2_in_rn,
    input[5:0] rd_in_rn,
    input[5:0] rd2_in_rn,

    //# {{control|Decoded Instruction Control}}
    output reg sc_ready,

    //# {{control|Register Control}}
    input[5:0] reg1_finished,
    input[5:0] reg2_finished,

    //# {{data|Execution Units Register Data}}
    output reg[5:0] rd_out_rn,
    output reg[5:0] rd2_out_rn,

    //# {{control|Execution Units Control Signals}}
    output reg alu1_en,
    output reg alu2_en,
    output reg advint_en,
    output reg memunit_en,
    output reg branch_en,

    input alu1_busy,
    input alu2_busy,
    input advint_busy,
    input memunit_busy,
    input branch_busy
    );

    reg[63:0] reg_busy;

    wire alu_type, advint_type, memunit_type, branch_type;
    assign alu_type = ~unit[2];
    assign advint_type = ~type && unit==3'h4;
    assign memunit_type = type && (unit==3'h4 | unit==3'h5 | unit==3'h6);
    assign branch_type = unit==3'h7;

    wire instIssued;
    assign instIssued = alu1_en | alu2_en | advint_en | memunit_en | branch_en;

    reg operand_unavailable;

    always @(*)
    begin
        operand_unavailable = 0;

        //The register was previously busy
```

(continues on next page)

(continued from previous page)

```
60          if(reg_busy[r1_in_rn] && r1_in_rn!=reg1_finished && r1_in_rn!=reg2_finished)␣
    ↪operand_unavailable = 1;
61          else if(reg_busy[r2_in_rn] && r2_in_rn!=reg2_finished && r2_in_rn!=reg1_
    ↪finished) operand_unavailable = 1;
62
63          //We just issued something to an execution unit
64          else if(instIssued) begin
65              //The incoming source register is non-zero
66              if(|r1_in_rn) begin
67                  //And it matches the previous destination register number.  We
68                  //will stall here until it is picked up by reg_busy next cycle
69                  if(rd_out_rn==r1_in_rn) operand_unavailable = 1;
70                  else if(rd2_out_rn==r1_in_rn) operand_unavailable = 1;
71
72              end
73              if(|r2_in_rn) begin
74                  if(rd_out_rn==r2_in_rn) operand_unavailable = 1;
75                  else if(rd2_out_rn==r2_in_rn) operand_unavailable = 1;
76              end
77          end
78      end
79
80      always @(*)
81      begin
82          sc_ready = 0;
83          if(~operand_unavailable & ~branch_busy) begin //TODO Branch busy condition␣
    ↪can probably be moved outside the module via muxing of the unit_en lines with the␣
    ↪operation cancel signal
84              if(alu_type & (~alu1_busy | ~alu2_busy)) sc_ready = 1;
85              else if(advint_type & ~advint_busy) sc_ready = 1;
86              else if(memunit_type & ~memunit_busy) sc_ready = 1;
87              else if(branch_type) sc_ready = 1;
88          end
89      end
90
91      always @(posedge clk or negedge rst_n)
92      begin
93          if(~rst_n) begin
94              alu1_en <= 0;
95              alu2_en <= 0;
96              advint_en <= 0;
97              memunit_en <= 0;
98              branch_en <= 0;
99              rd_out_rn <= 6'h0;
100             rd2_out_rn <= 6'h0;
101             reg_busy <= 64'h0;
102
103         end else begin
104             //Only allow the scheduling of instructions if the source registers
105             //aren't the destination of in-progress instructions.
106             alu1_en <= 0;
107             alu2_en <= 0;
108             advint_en <= 0;
109             memunit_en <= 0;
110             branch_en <= 0;
111
112             reg_busy[reg1_finished] <= 0;
```

(continues on next page)

(continued from previous page)

```
113             reg_busy[reg2_finished] <= 0;
114
115             if(~operand_unavailable & ~branch_busy) begin
116                 if(alu_type & ~alu1_busy) begin
117                     alu1_en <= 1;
118                     rd_out_rn <= rd_in_rn;
119                     if(|rd_in_rn) reg_busy[rd_in_rn] <= 1;
120
121                 end else if(alu_type & ~alu2_busy) begin
122                     alu2_en <= 1;
123                     rd_out_rn <= rd_in_rn;
124                     if(|rd_in_rn) reg_busy[rd_in_rn] <= 1;
125
126                 end else if(advint_type & ~advint_busy) begin
127                     advint_en <= 1;
128                     rd_out_rn <= rd_in_rn;
129                     rd2_out_rn <= rd2_in_rn;
130                     if(|rd_in_rn) reg_busy[rd_in_rn] <= 1;
131                     if(|rd2_in_rn) reg_busy[rd2_in_rn] <= 1;
132
133                 end else if(memunit_type & ~memunit_busy) begin
134                     memunit_en <= 1;
135                     rd_out_rn <= rd_in_rn;
136                     if(|rd_in_rn && unit!=3'h6) reg_busy[rd_in_rn] <= 1;
137
138                 end else if(branch_type) begin
139                     branch_en <= 1;
140                     rd_out_rn <= rd_in_rn;
141                     //No need to mark R63 busy. If the branch is taken, other
142                     //instructions in the pipeline are cancelled.
143                 end
144             end
145         end
146     end
147
148 endmodule
```

CHAPTER

SIX

FOOTNOTES