Jolt Formal Specification

Cat

February 02, 2026

1. Expanded Instructions

RISC-V descriptions are taken from https://msyksphinz-self.github.io/riscv-isadoc/

1.1. MULH

Done above

1.2. DIV (Special)

Prover: computes q and r (fills advice fields) Prover: constructs aproof that these values satisfy all constraints Verifier: checks the proof

```asm
div rd,rs1,rs2
```

Description perform an XLEN bits by XLEN bits signed integer division of rs1 by rs2, rounding towards zero.

```rust
x[rd] = x[rs1] /s x[rs2]
```

1.2.a. Virutal Instructions

```rust
let a0 = self.operands.rs1; // dividend
let a1 = self.operands.rs2; // divisor
let a2 = allocator.allocate(); // quotient (from oracle)
let a3 = allocator.allocate(); // |remainder| (from oracle)
let t0 = allocator.allocate(); // adjusted divisor
let t1 = allocator.allocate(); // temporary
let shmat = 63

asm.emit_j::<VirtualAdvice>(*a2, 0); // quotient
asm.emit_j::<VirtualAdvice>(*a3, 0); // |remainder|

// Handle special cases: div-by-zero and overflow
// If divisor 0, check q = u64::MAX, otherwise do nothing
// rs1 = a1 = divisor
// rs2 = a2 = quotient
asm.emit_b::<VirtualAssertValidDiv0>(a1, *a2, 0);
// t0 = either is divisor (a1) at or 1 (based on overflow)
asm.emit_r::<VirtualChangeDivisor>(*t0, a0, a1);

// Verify no overflow: quotient × divisor must not overflow
```

```
// t1 = High64[q * t0]
asm.emit_r::<MULH>(*t1, *a2, *t0); // High bits of multiplication

let t2 = allocator.allocate();
let t3 = allocator.allocate();

// Take lower 64 bits of q*t0 into t2 (always fits in 64 bits)
asm.emit_r::<MUL>(*t2, *a2, *t0); // quotient × adjusted_divisor
// shmat is 63 - so lower 5 bits are all 1
// t3 = sign(t2) = t2>>63
asm.emit_i::<SRAI>(*t3, *t2, shmat); // Sign-extend low bits
// Check that t1 = t3 or High[q*t0] = t2 >> 63 ??
asm.emit_b::<VirtualAssertEQ>(*t1, *t3, 0); // Assert no overflow

// Apply sign of dividend to remainder
asm.emit_i::<SRAI>(*t1, a0, shmat); // Sign bit of dividend
asm.emit_r::<XOR>(*t3, *a3, *t1); // XOR with |remainder|
asm.emit_r::<SUB>(*t3, *t3, *t1); // Two's complement if negative

// Verify: dividend = quotient × divisor + remainder
asm.emit_r::<ADD>(*t2, *t2, *t3); // Add signed remainder
asm.emit_b::<VirtualAssertEQ>(*t2, a0, 0); // Assert equals dividend

// Verify: |remainder| < |divisor|
asm.emit_i::<SRAI>(*t1, *t0, shmat); // Sign bit of adjusted divisor
asm.emit_r::<XOR>(*t3, *t0, *t1); // XOR to get magnitude
asm.emit_r::<SUB>(*t3, *t3, *t1); // |adjusted_divisor|
// checks if (divisor) `t3= 0` or (remainder) `a3 < t3`
asm.emit_b::<VirtualAssertValidUnsignedRemainder>(*a3, *t3, 0);

// Move quotient to destination register
asm.emit_i::<ADDI>(self.operands.rd, *a2, 0);
```

Claim:

Lemma 1.2.1.1: The assertion $\text{High64}[q \cdot t_0] = (\text{Low64}[q \cdot t_0] \gg 63)$ passes if and only if $q \cdot t_0 \in [-2^{63}, 2^{63} - 1]$.

Proof. $\Longrightarrow$:

Suppose $q \cdot t_0 \in [-2^{63}, 2^{63} - 1]$. When viewed as a 128-bit value, this equals the sign extension of its 64-bit representation.

By definition of sign extension:

$$\text{High64} = \begin{cases} \texttt{0x0000\_0000\_0000\_0000} \text{ if } q \cdot t_0 \geq 0 \\ \texttt{0xFFFF\_FFFF\_FFFF\_FFFF} \text{ if } q \cdot t_0 < 0 \end{cases} \tag{1}$$

The arithmetic right shift by 63 positions produces:

$$\text{Low64} \gg 63 = \begin{cases} 0 & \text{if sign bit of Low64 is 0} \\ -1 = \texttt{0xFFFF\_FFFF\_FFFF\_FFFF} & \text{if sign bit of Low64 is 1} \end{cases} \tag{2}$$

Since the sign bit of Low64 correctly represents whether $q \cdot t_0 \geq 0$ or $q \cdot t_0 < 0$, we have:

$$\text{High64} = \text{Low64} \gg 63 \tag{3}$$

Therefore the assertion passes.

$\Longleftarrow$ If the assertion passes, then $q \cdot t_0 \in [-2^{63}, 2^{63} - 1]$

Suppose the assertion $\text{High64} = \text{Low64} \gg 63$ passes.

Case 1: $\text{Low64} \geq 0$ (when interpreted as signed 64-bit, i.e., sign bit = 0)
- Then $\text{Low64} \gg 63 = 0$
- So $\text{High64} = 0 = \texttt{0x0000\_0000\_0000\_0000}$
- The 128-bit value is: $\texttt{0x0000\_0000\_0000\_0000} \parallel \text{Low64}$
- This represents a value in $[0, 2^{63} - 1]$ ✓

Case 2: $\text{Low64} < 0$ (when interpreted as signed 64-bit, i.e., sign bit = 1)
- Then $\text{Low64} \gg 63 = -1 = \texttt{0xFFFF\_FFFF\_FFFF\_FFFF}$
- So $\text{High64} = \texttt{0xFFFF\_FFFF\_FFFF\_FFFF}$
- The 128-bit value is: $\texttt{0xFFFF\_FFFF\_FFFF\_FFFF} \parallel \text{Low64}$
- This represents Low64 sign-extended to 128 bits
- Since $\text{Low64} \in [-2^{63}, -1]$ as a signed 64-bit value, the 128-bit representation is also in $[-2^{63}, -1]$ ✓

□

Lemma 1.2.1.2: $t_3 = \text{sign}(a)|r|$

Proof. **Step 1:** Extract sign of dividend

$$t_1 = a_0 \gg 63 = \begin{cases} 0 = \texttt{0x0000\_0000\_0000\_0000} & \text{if } a \geq 0 \\ -1 = \texttt{0xFFFF\_FFFF\_FFFF\_FFFF} & \text{if } a < 0 \end{cases} \tag{4}$$

Step 2: XOR with absolute remainder

$$t_3 = a_3 \oplus t_1 \tag{5}$$

Step 3: Subtract to apply two's complement

$$t_3 = t_3 - t_1 \tag{6}$$

Case 1: $a = a_0 \geq 0$ (so $r = a_3$ should be non-negative)

- $t_1 = 0$

- $t_3 = a_3 \oplus 0 = a_3$
- $t_3 = a_3 - 0 = a_3$
- Result: $t_3 = |r| = r$

Case 2: $a < 0$ (so $r = a_3$ should be ≤ 0)

- $t_1 = -1 =$ `0xFFFF_FFFF_FFFF_FFFF`
- $t_3 = a_3 \oplus$ `0xFFFF_FFFF_FFFF_FFFF` $= \tilde{a}_3$ (bitwise NOT)
- $t_3 = \tilde{a}_3 - (-1) = \tilde{a}_3 + 1$
- By two's complement: $\tilde{a_3} + 1 = -a_3 = -|r|$
- Result: $t_3 = -|r|$

□

1.2.b. The Code (last two conditons are quite easy)

```rust
// Verify: dividend = quotient × divisor + remainder
asm.emit_r::<ADD>(*t2, *t2, *t3);     // t2 = q·b + r
asm.emit_b::<VirtualAssertEQ>(*t2, a0, 0); // Assert equals dividend
```

1.2.c. What This Checks

At this point in the code:

- `t2` $= q \cdot t_0$ where t_0 is the adjusted divisor
- `t3` $= r$ with correct sign
- `a0` $= a$ (dividend)

The assertion verifies: $a = q \cdot b + r$

1.2.d. Proof of Correctness

Normal case: $t_0 = b$ (no overflow)

- The code computes: $t_2 + t_3 = q \cdot b + r$
- The assertion checks: $q \cdot b + r = a$
- This is exactly the division algorithm requirement ✓

Overflow case: $(a, b) = (-2^{63}, -1)$, and $r = 0$ from the next condition that $|r| < |b'|$ (Adjusted)

- To avoid computing $q \cdot b = (-2^{63}) \cdot (-1) = 2^{63}$ (overflow)
- The code sets $t_0 = 1$ (adjusted divisor)
- Then the check becomes $q + r = (-2^{63}) \cdot 1 = -2^{63}$
- If the prover was honest $r = 0$ and $q = -2^{63}$ is the correct answer, and passes. The assertion verifies: $q \cdot t_0 + r = -2^{63} + 0 = a$ ✓

Theorem 1.2.4.1: State matches.

Proof. All we need to show is that the final value in `rd` which we denote with z is actually $\frac{x}{y}$ signed integer division with rounding towards 0. We start with first two

instructions which simply stores in virtual registers `a2` and `a3` claimed quotient q, and remainder r.

Next, we focus on the last instruction which simply returns q as the answer. The thing to show here, if the prover uses a bad value for q and r, one that is not what the real RISC-V cpu would have computed, we would have a panic, and thus the program would crash. □

⚠ Warning

Note for ARI:

The guest program says div; and then i expand that to above. Now notice that the program starts with two instrucions.

```rust
asm.emit_j::<VirtualAdvice>(*a2, 0); // quotient
asm.emit_j::<VirtualAdvice>(*a3, 0); // |remainder|
```

These two instructions are telling the CPU, write into virtual registers `a2` and `a3` the quotient and remainder. Now where does the CPU, get these answers from? Well it gets it from somewhere, as q and r which it then proceeds to put into these registers. The code below shows what the CPU actually does during tracing.

```rust
let mut inline_sequence = self.inline_sequence(&cpu.vr_allocator, cpu.xlen);
// The first instructioon
if let Instruction::VirtualAdvice(instr) = &mut inline_sequence[0] {
  instr.advice = quotient;
} else {
  panic!("Expected Advice instruction");
}
// The Second instruction
if let Instruction::VirtualAdvice(instr) = &mut inline_sequence[1] {
  instr.advice = remainder;
} else {
  panic!("Expected Advice instruction");
}

// With these values in there it just executes as normal.
let mut trace = trace;
for instr in inline_sequence {
  instr.trace(cpu, trace.as_deref_mut());
}
```

1.3. SUBW

```asm
subw rd,rs1,rs2
```

Description: Subtract the 32-bit of registers rs1 and 32-bit of register rs2 and stores the result in rd. Arithmetic overflow is ignored and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register

1.3.a. Virtual Sequence

```rust
asm.emit_r::<SUB>(self.operands.rd, self.operands.rs1,
self.operands.rs2);
asm.emit_i::<VirtualSignExtendWord>(self.operands.rd, self.operands.rd,
0);
```

where the instructions `SUB` and `VirtualSignExtendWord` are defined as follows:

`SUB`: Subs the register rs2 from rs1 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

```rust
//Implementation
x[rd] = x[rs1] - x[rs2]
```

`VirtualSignExtendWord`: Sign-extends the lower 32 bits of a register to 64 bits:

```rust
//Implementation
(cpu.x[self.operands.rs1 as usize] << 32) >> 32
```

Theorem 1.3.1.1: Let the triple `(rd, rs1, rs2)` denote the machine state. Then machine state before and after execution of `subw` and the virtual sequence is identical.

Proof. `rs1, rs2` remain unchanged before and after the instruction. All there is to show is that the value in `rd` is the same.

As defined earlier let x, y, z denote the `w` bit values in `rs1, rs2, rd` represented in unsigned form. Similarly, x', y', z' are the same but in signed representation.

The first instruction gives us :

$$z = x - y \pmod{2^{64}} \tag{7}$$

By construction, the lower 32 bits of z are exactly with arithmetic overflow ignored.

$$z[31:0] = (x[31:0] - y[31:0]) \bmod 2^{32} \tag{8}$$

Finally we sign extend the lower 32 bits of z to get the desired answer. □

1.4. SLLI

```asm
slli rd,rs1,shamt
```

Performs logical left shift on the value in register `rs1` by the shift amount held in the lower 5 bits of the immediate. In RV64, bit-25 is used to shamt[5].

1.4.a. Virtual Sequence

```rust
shift = immm & 0x3f; // Lower 6 bits
asm.emit_i::<VirtualMULI>(self.operands.rd, self.operands.rs1, 1 << shift);
```

Theorem 1.4.1.1: States mactch

Proof.

Let variable s denote the `shift` amount stored in `Imm[0:5]` and x denote the value in `rs1`. It's a mathematical fact that

$$z = x \times 2^s = x \ll s \tag{9}$$

where $\times$ denotes wrapping multiplication i.e we drop the overflow. Therefore, the final value in `rd` is correct. □

1.5. SRLI

```asm
rd, rs1, immm
```

Description: Performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

1.5.a. Virtual Sequence

```rust
shift = imm & 0x3f // shift \in [0, 63]
len = 64
ones = (1u128 << (len - shift)) - 1;
bitmask = (ones << shift) as u64
// Virtual SRLI right shifts contents of rs1 by the number
// of trailing zeros in the bitmask
asm.emit_vshift_i::<VirtualSRLI>(self.operands.rd, self.operands.rs1, bitmask);
```

Theorem 1.5.1.1: Match

Proof. Let i denote the immediate value in the instruction. Define $w = 64$ and $s = i \wedge 0\text{x}3\text{F}$ be the number of bits to shift.

From the definition of `srli` we have that the destination register will contain

$$z = x \gg s \tag{10}$$

Define $o := 2^{w-s} - 1$. We claim o has s 0's in it's higher order bits. If this claim were to be true, then the bitmask b created by loggically shifting o by s is given by $b := o \ll s$, and would would have exactly s trailing 0's. We know that `VirtualSRLI` right shifts contents of `rs1` by number of trailing 0's of `bitmask`. This gives us that $z = x \gg s$.

All that's left to show is that b has s 0's in its higher order bits. For any $k \geq 0$, 2^k has bit k set to 1, and k trailing 0's, and $w - k - 1$ prefix bits are 0. This gives us $2^k - 1$, clears the k'th bit, and sets all trailing 0's to ones, giving us $w - k$ prefix bit 0's. Now set $k = w - s$, we get our result.

□

1.6. LW

asm

```
lw rd,offset(rs1)
```

Description: Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

rust

```
x[rd] = sext(M[x[rs1] + sext(offset)][31:0])
```

1.6.a. Virtual Sequence

rust

```
// Check if x[rs1] + imm % 4 == 0 panic otherwise
asm.emit_halign::<VirtualAssertWordAlignment>(self.operands.rs1,
self.operands.imm);
// v_address = x[rs1] + imm[0:11] with wrapping add (this is the address
from which i want to load)
asm.emit_i::<ADDI>(*v_address, self.operands.rs1, self.operands.imm as
u64);
// v_dword_address = v_address & (-8 as i64) as u64
asm.emit_i::<ANDI>(*v_dword_address, *v_address, -8i64 as u64);
// v_dword = M[v_dword_address]; which is v_address with lowest 3 bits
cleared
asm.emit_ld::<LD>(*v_dword, *v_dword_address, 0);
// v_shift = v_address << 3
asm.emit_i::<SLLI>(*v_shift, *v_address, 3);
// rd = v_d >> v_shift[0:4] ; r-shift x[rs1] amount held in the lower 5
bits of v_shift
asm.emit_r::<SRL>(self.operands.rd, *v_dword, *v_shift);
// rd = sign-extend(rd)
asm.emit_i::<VirtualSignExtendWord>(self.operands.rd, self.operands.rd,
0);
```

Note for Ari: this proof can be shortened, but i overgenerated so i remember.

Proof.

```
Address (hex)    LB   LH   LW   LD
0x1000           ✅   ✅   ✅   ✅ <- this is now v_(dwa)
0x1001           ✅   ❌   ❌   ❌
0x1002           ✅   ✅   ❌   ❌
0x1004           ✅   ✅   ✅   ❌ <- pretend this was v_a = v_(wa)
0x1008           ✅   ✅   ✅   ✅
```

As we will load 32 bits or 1 word into memory, we need our address to be word aligned i.e divisible by 4. This is the first check.

1. We first compute the address $a := x + i$, where i=`imm`, and check that $a \bmod 4 = 0$.
2. If the above test passes set $v_{wa} := a$ where a is defined above.
3. Clear the lowest 3 bits of v_{wa} and set $v_{dwa} = v_{(wa)'} \parallel 000$ where $v_{(wa)'}$ is the highest $w - 3$ bits of v_{wa}.

We do this because we need our temporary double word address to be divisible by 8, or double-word aligned.

4. Our double word aligned adress is good for a load; $v_d := M[v_{dwa}]$; load 64 bits (2 words) from address in register v_{dwa}
5. $v_s := v_a \ll 3$

This is subtle. Note that we are guanrateed that v_a is word-aligned, but it could also be double-word aligned. In v_d we have the 64 bits read from location v_a. If v_a were also double word aligned, then it would mean that $v_a = v_{wa} = v_{dwa}$. So we want the lower 32 bits of v_d as the answer. Instead, if we v_a was not double word aligned, then we want the high 32 bits of v_d as the answer (as shown in the example above).

The way `SRL` works is it will look at the lower 5 bits of v_s to decide how to much to shift right by. If we are in the first case, and v_a is double word aligned, then it already has 3 trailing 0's. Now v_s will have 6 trailing 0's, and we are safe that we do not right shift v_d at all. We just output the lower 32 bits sign extended as the answer. If v_a were not double word aligned, then we are guranteed that there should be a 1 in index 2. Now if we were to left shift 3 units, we'd have $v_s = 32$, and this gives us the higher 32 bits. This completes the proof.

□

1.7. SRA

```asm
sra rd,rs1,rs2
```

Description Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2

Implementaion:

```rust
```

```
x[rd] = x[rs1] >>s x[rs2]
```

1.7.a. Virtualisation

```rust
// v_bitmask = bitmask such that it has s=x[rs2][0:5] trailing 0s
asm.emit_i::<VirtualShiftRightBitmask>(*v_bitmask, self.operands.rs2, 0);
// Right shift x[rs1] by num trailing 0's in v_bitmask and store in rd
asm.emit_vshift_r::<VirtualSRA>(self.operands.rd, self.operands.rs1,
*v_bitmask);
```

> *Proof.*
>
> 1. The first instruction constructs v_b with s traililing 0's, where $s = y[0:5]$.
> 2. Now we right shift x by s
>
> The equivalence holds trivially. □

1.8. AMONEXW

```asm
amomaxu.w rd,rs2,(rs1)
```

Description Atomically load a 32-bit unsigned data value from the address in rs1, place the value into register rd, apply unsigned max the loaded value and the original 32-bit unsigned value in rs2, then store the result back to the address in rs1.

```rust
x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])
```

1.8.a. Virtual Instructions

```rust

asm.emit_halign::<VirtualAssertWordAlignment>(rs1, 0);
// Use v_shift temporarily to hold aligned address
asm.emit_i::<ANDI>(v_shift, rs1, -8i64 as u64);
asm.emit_ld::<LD>(v_dword, v_shift, 0);
// Now compute the actual shift value
asm.emit_i::<SLLI>(v_shift, rs1, 3);
asm.emit_r::<SRL>(v_rd, v_dword, v_shift);


asm.emit_i::<VirtualZeroExtendWord>(*v_rs2, self.operands.rs2, 0);
// Zero-extend v_rd in place into v_sel_rd (temporarily)
asm.emit_i::<VirtualZeroExtendWord>(*v_sel_rd, *v_rd, 0);
// Compare: v_sel_rd (zero-extended v_rd) < v_rs2
asm.emit_r::<SLTU>(*v_sel_rs2, *v_sel_rd, *v_rs2);
// Invert selector to get selector for v_rd
asm.emit_i::<XORI>(*v_sel_rd, *v_sel_rs2, 1);
// Select maximum using multiplication
asm.emit_r::<MUL>(*v_rs2, *v_sel_rs2, self.operands.rs2);
asm.emit_r::<MUL>(*v_sel_rd, *v_sel_rd, *v_rd);
```

```
asm.emit_r::<ADD>(*v_rs2, *v_sel_rd, *v_rs2);


asm.emit_i::<ORI>(v_mask, 0, -1i64 as u64);
asm.emit_i::<SRLI>(v_mask, v_mask, 32);
asm.emit_r::<SLL>(v_mask, v_mask, v_shift);
// Use v_shift as temporary after it's been used for shifting
asm.emit_r::<SLL>(v_shift, rs2, v_shift);
asm.emit_r::<XOR>(v_shift, v_dword, v_shift);
asm.emit_r::<AND>(v_shift, v_shift, v_mask);
asm.emit_r::<XOR>(v_dword, v_dword, v_shift);
// Recompute aligned address for store
asm.emit_i::<ANDI>(v_mask, rs1, -8i64 as u64);
asm.emit_s::<SD>(v_mask, v_dword, 0);
asm.emit_i::<VirtualSignExtendWord>(rd, v_rd, 0);
```

1.9. SRLIW

asm

```
slliw rd,rs1,shamt
```

Description Performs logical left shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with imm[5] $\neq 0$ are reserved.

The virtual instructions are as follows:

rust

```
asm.emit_i::<SLLI>(*v_rs1, self.operands.rs1, 32);
(shift, len) = ((self.operands.imm & 0x1f) + 32, 64),
let ones = (1u128 << (len - shift)) - 1;
let bitmask = (ones << shift) as u64;
asm.emit_vshift_i::<VirtualSRLI>(self.operands.rd, *v_rs1, bitmask);
asm.emit_i::<VirtualSignExtendWord>(self.operands.rd, self.operands.rd,
  0);
```

Theorem 1.9.1: Match

Proof. Want to show that the above virtual instructions performs `rd = (rs1 as u32) >> shamt` where s=`shamt = imm[4:0]`

Let x, z denote the values in `rs1, rd` respectively. Let v_1 denote the contents of the virtual register `v_rs1`. Based in the first instcution we have v_1 holds the lower 32 bits of x in its upper 32 bits.

$$v_1 = x[0:31] \parallel 0 \tag{11}$$

Define $s' := s + 32$ where s is the specified shift amount. This implies $s' \in [32, 63]$, as $s \in [0, 31]$.

Define

$$o := 2^{64-(s+32)} - 1 = 2^{32-s} - 1 \tag{12}$$

Here o is a w bit integer with exactly $(32 - s)$ lower bits set to 1. Therefore the $32 + s$ upper bits are set to 0.

$$b := (o \ll s') = (o \ll (s + 32)) \tag{13}$$

Now b has $s + 32$ trailing 0's so when we right shift v_1 by $s + 32$ bits, the 32 first ensures that $v_1 := 0 \parallel x[0 : 31]$, so it's x as `u32`, and then it right shifts by s bits as desired. □

1.10. SRL

```asm
1 srl rd, rs1, rs2
```

Description: Logical right shift the contents of `rs1` by `shamt`, where `shamt` lower 6 bits of `rs2` as a unsigned integer.

1.10.a. Virtual Sequence

```rust
1 asm.emit_i::<VirtualShiftRightBitmask>(*v_bitmask, self.operands.rs2, 0);
2 asm.emit_vshift_r::<VirtualSRL>(self.operands.rd, self.operands.rs1,
  *v_bitmask);
```

Theorem 1.10.1.1: Match

Proof.

1. Let s denote the lower 6 bits of `rs2`. The first instruction sets `v_bitmask` to w bit number with s trailing 0's, from the guarantees of `VirtualShiftRightBitmask`.
2. The second instruction logically right shifts the contents of `rs1` by s (the number of trailing 0's in `v_bitmask`).

□

1.11. SW

Store at memory location in `rs1` the lower 32 bits of the contents of `rs2`.

```rust
1
2 asm.emit_halign::<VirtualAssertWordAlignment>(self.operands.rs1,
  self.operands.imm);
3 asm.emit_i::<ADDI>(*v_address, self.operands.rs1, self.operands.imm as
  u64);
4 asm.emit_i::<ANDI>(*v_dword_address, *v_address, -8i64 as u64);
5 asm.emit_ld::<LD>(*v_dword, *v_dword_address, 0);
6 asm.emit_i::<SLLI>(*v_shift, *v_address, 3);
```

```
 7 asm.emit_i::<ORI>(*v_mask, 0, -1i64 as u64);
 8 asm.emit_i::<SRLI>(*v_mask, *v_mask, 32);
 9 asm.emit_r::<SLL>(*v_mask, *v_mask, *v_shift);
10 asm.emit_r::<SLL>(*v_word, self.operands.rs2, *v_shift);
11 asm.emit_r::<XOR>(*v_word, *v_dword, *v_word);
12 asm.emit_r::<AND>(*v_word, *v_word, *v_mask);
13 asm.emit_r::<XOR>(*v_dword, *v_dword, *v_word);
14 asm.emit_s::<SD>(*v_dword_address, *v_dword, 0);
```

Theorem 1.11.1: Match

Proof.

We first check if `v_address` is word aligned or not. If not we panic immediately, as the original specification would. If it were to be aligned we have two cases:

1. `v_address` is also double word aligned i.e divisible by 8.
2. `v_address` is only single world aligned but not double word aligned i.e. not divisible by 8.

`asm.emit_i::<ANDI>(*v_dword_address, *v_address, -8i64 as u64);` `v_dword_address` is by definition double word aligned as we AND it with a mask with 3 trailing 0s.

`asm.emit_ld::<LD>(*v_dword, *v_dword_address, 0);` `v_dword` stores 64 bits starting at address `v_dword_address`

rust
```
1 asm.emit_i::<SLLI>(*v_shift, *v_address, 3);
```

Based on whether we are in case 1 or case 2, `v_shift=0` or `v_shift=32`

rust
```
1 asm.emit_i::<ORI>(*v_mask, 0, -1i64 as u64);
```

This sets `v_mask` to the all ones bits string.

`asm.emit_i::<SRLI>(*v_mask, *v_mask, 32);` This clears the upper 32 bits of the `mask`.

`asm.emit_r::<SLL>(*v_mask, *v_mask, *v_shift);` Now if we are in case 1, `v_mask= 0 || 1` were upper 32 bits are 0, and lower 32 bits are 1. If we are in case 2, `v_mask 1 || 0` were upper 32 bits are 1, and lower 32 bits are 0.

`asm.emit_r::<SLL>(*v_word, self.operands.rs2, *v_shift);` If we are in case 1, then `v_word = x[rs2]` If we are in case 2, then `v_word = x[rs2][0:31] || 0`, i.e the lower 32 bits of `rs2` are the upper 32 bits of `v_word`.

Let `mem` = `v_dword`.

`asm.emit_r::<XOR>(*v_word, *v_dword, *v_word);` In case 1, `v_word = x[rs2] XOR mem` In case 2, `v_word = mem[63:32] || (mem[0:31] XOR x[rs2][0:31])`

`asm.emit_r::<AND>(*v_word, *v_word, *v_mask);` If case 1, where `v_word` was the entire contents of `rs2`, and `v_mask= 0 ||1`, this will set `v_word` to just `0 || x[rs2][0:31] XOR mem[0:31]`.

If case 2, where `v_Word = x[rs2][0:31] XOR mem[63:32] || 0`, and `v_mask = 1 || 0`, this will keep `v_word` as is.

`asm.emit_r::<XOR>(*v_dword, *v_dword, *v_word);`

Now in case 1, `v_dword = mem[63:32] || x[rs2][31:0] XOR mem[0:31] XOR mem[31:0] = mem[63:32] || x[rs2][31:0]`, exactly what I want. By similar, analysis, In case 2, `v_dword = x[rs2][31:0] || mem[31:0]`.

Store it in the right location, and that completes the proof. `asm.emit_s::<SD>(*v_dword_address, *v_dword, 0);`

□

1.12. REMU

If the quotient and remainder are correct, then we want to show all the checks will pass and the final value in `rd = a3`.

Let `r` be the remainder the native RISC-V instruction `REMU` returns. We are guaranteed that there exists some `q` such that `a0 = a1 x q + r` where `0 <= r < a1` (folklore number theory fact). Assume that the prover sets `a2=q` and `a3=r`.

Then, `a2 x a1` should not overflow, as $a0 = a1 \times a2 + a3$ and `a0` $\in [0, 2^{64} - 1]$, then both the following check passes.

```rust
1 asm.emit_b::<VirtualAssertMulUNoOverflow>(*a2, a1, 0)
2 asm.emit_b::<VirtualAssertEQ>(*t1, a0, 0)
```

`asm.emit_b::<VirtualAssertLTE>(*t0, *t1, 0)` checks `a1 x a2 <= a1 x a2 + a3` which passes as `r=a3 >= 0`,

`asm.emit_b::<VirtualAssertLTE>(*a3, *t1, 0)` checks if `a3 <= a1 x a2 + a3` which also holds with equality. All tests pass and `rd` is correct. The contrapositive, says that if one of the tests fails, then either `a2` and `a3` are different from `q` and `r`.

1.13. AMOSWAPW

```asm
1 TODO
```

Description: Load a word from `address`, where `address = x[rs1] + offset`, sign extend it and put it in `rd`. Also put the lower 32 bits of the contents of `rs2` in at the memory location `address`.

1.13.a. Virtual Sequence

```rust
asm.emit_halign::<VirtualAssertWordAlignment>(self.operands.rs1, 0);
asm.emit_i::<ANDI>(*v_shift, self.operands.rs1, -8i64 as u64);
asm.emit_ld::<LD>(*v_dword, *v_shift, 0);
asm.emit_i::<SLLI>(*v_shift, self.operands.rs1, 3);
asm.emit_r::<SRL>(*v_rd, *v_dword, *v_shift);
asm.emit_i::<ORI>(*v_mask, 0, -1i64 as u64);
asm.emit_i::<SRLI>(*v_mask, *v_mask, 32);
asm.emit_r::<SLL>(*v_mask, *v_mask, *v_shift);
asm.emit_r::<SLL>(*v_shift, self.operands.rs2, *v_shift);
asm.emit_r::<XOR>(*v_shift, *v_dword, *v_shift);
asm.emit_r::<AND>(*v_shift, *v_shift, *v_mask);
asm.emit_r::<XOR>(*v_dword, *v_dword, *v_shift);
asm.emit_i::<ANDI>(*v_mask, self.operands.rs1, -8i64 as u64);
asm.emit_s::<SD>(*v_mask, *v_dword, 0);
asm.emit_i::<VirtualSignExtendWord>(self.operands.rd, *v_rd, 0);
```

1.13.b. Proof Of Correctness

Proof on whiteboard.

1.14. amoord

Description: Atomically loads a 64-bit doubleword from the address in `rs1`, ORs it with `rs2`, stores the result back at the same address. Place the original value loaded from the address value in `rd`.

1.14.a. Virtual Sequence

```rust
asm.emit_ld::<LD>(*v_rd, self.operands.rs1, 0);
asm.emit_r::<OR>(*v_rs2, *v_rd, self.operands.rs2);
asm.emit_s::<SD>(self.operands.rs1, *v_rs2, 0);
asm.emit_i::<ADDI>(self.operands.rd, *v_rd, 0);
```

1.14.b. Proof Of Correctness

Lines 1-2 set `v_rs2 = (M[x[rs1]] OR x[rs2])` and Line 3 stores `v_rs2` at memory location stored in `rs1`. Line 4 puts `M[x[rs1]]` into `rd`.

1.15. LBU

Description: Loads an 8-bit byte from memory at `rs1 + offset`, zero-extending the result into `rd`.

1.15.a. Virtual Sequence

```rust
asm.emit_i::<ADDI>(*v_address, self.operands.rs1, self.operands.imm as
  u64);
asm.emit_i::<ANDI>(*v_dword_address, *v_address, -8i64 as u64);
asm.emit_ld::<LD>(*v_dword, *v_dword_address, 0);
```

```
4 asm.emit_i::<XORI>(*v_shift, *v_address, 7);
5 asm.emit_i::<SLLI>(*v_shift, *v_shift, 3);
6 asm.emit_r::<SLL>(self.operands.rd, *v_dword, *v_shift);
7 asm.emit_i::<SRLI>(self.operands.rd, self.operands.rd, 56);
```

1.15.b. Proof Of Correctness

1.16. SLLW

Description: Shifts the lower 32 bits of `rs1` left by the amount in lower 5 bits of `rs2`, sign-extending the 32-bit result into `rd`.

1.16.a. Virtual Sequence

rust

```
1
2 asm.emit_i::<VirtualPow2W>(*v_pow2, self.operands.rs2, 0);
3 asm.emit_r::<MUL>(self.operands.rd, self.operands.rs1, *v_pow2);
4 asm.emit_i::<VirtualSignExtendWord>(self.operands.rd, self.operands.rd,
  0);
```

1.16.b. Proof Of Correctness

Left shift is by definition multuplication by a power of 2. The lowest 5 bits of `rs2` should be a number between 0 and 31.

By the guarantees of `VirtualPow2W`,

rust

```
1 v_pow2 = 1 << (x[rs2] % 32) = 2^{x[rs2]}
```

Line 2, ensures `rd = v_pow2 x x[rs1]`, where `mul` Multiplies `rs1` by `rs2` and stores the lower `XLEN` bits of the product in `rd`. Then Line 3, sign-extends the lower 32 bits to 64 bits by the guarantees of `VirtualSignExtendWord`.

1.17. LH

Description: Loads a 16-bit halfword from memory at `rs1 + offset`, sign-extending the result into `rd`.

1.17.a. Virtual Sequence

rust

```
1 asm.emit_halign::<
2 VirtualAssertHalfwordAlignment,
3   >(self.operands.rs1, self.operands.imm);
4 asm.emit_i::<ADDI>(*v_address, self.operands.rs1, self.operands.imm as
  u64);
5 asm.emit_i::<ANDI>(*v_dword_address, *v_address, -8i64 as u64);
6 asm.emit_ld::<LD>(*v_dword, *v_dword_address, 0);
7 asm.emit_i::<XORI>(*v_shift, *v_address, 6);
8 asm.emit_i::<SLLI>(*v_shift, *v_shift, 3);
9 asm.emit_r::<SLL>(self.operands.rd, *v_dword, *v_shift);
10 asm.emit_i::<SRAI>(self.operands.rd, self.operands.rd, 48);
```

1.17.b. Proof Of Correctness