RFC ls003 Big Integer

URLs:

- https://libre-soc.org/openpower/sv/biginteger/analysis/
- https://libre-soc.org/openpower/sv/rfc/ls003/
- https://bugs.libre-soc.org/show_bug.cgi?id=960
- https://git.openpower.foundation/isa/PowerISA/issues/91

Severity: Major

Status: New

Date: 20 Oct 2022 - v2 TODO

Target: v3.2B

Source: v3.0B

Books and Section affected: **UPDATE**

```
Book I 64-bit Fixed-Point Arithmetic Instructions 3.3.9.1
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Summary

Instructions added

```
maddedu - Multiply-Add Extended Double Unsigned
maddedus - Multiply-Add Extended Double Unsigned/Signed
divmod2du - Divide/Modulo Quad-Double Unsigned
dsld - Double Shift Left Doubleword
dsrd - Double Shift Right Doubleword
```

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

```
Addition of five new GPR-based instructions
```

Impact on software:

```
Requires support for new instructions in assembler, debuggers,
and related tools.
```

Keywords:

```
GPR, Big-integer, Double-word
```

Motivation

- Similar to `maddhdu` and `maddld`, but allow for a big-integer rolling accumulation affect: `RC` effectively becomes a 64-bit carry in chains of highly-efficient loop-unrolled arbitrary-length big-integer operations.
- Similar to `divdeu`, and has similar advantages to `maddedu`, Modulo result is available with the quotient in a single instruction allowing highly-efficient arbitrary-length big-integer division.
- Combining at least three instructions into one, the `dsld` and `dsrd` instructions make shifting an arbitrary-length big-integer vector by a scalar 64-bit quantity highly efficient.

Notes and Observations:

1. It is not practical to add Rc=1 variants when VA-Form is used and there is a **pair** of results produced.
2. An overflow variant (XER.OV set) of `divmod2du` would be valuable but VA-Form EXT004 is under severe pressure.
3. Both `maddhdu` and `divmod2du` instructions have been present in Intel x86 for several decades. Likewise, `dsld` and `dsrd`.
4. None of these instruction is present in VSX.
5. `maddedu` and `divmod2du` are full inverses of each other, including when used for arbitrary-length big-integer arithmetic.
6. These are all 3-in 2-out instructions. If Power ISA did not already have LD/ST-with-update instructions and instructions with `RAp` and `RTp` then these instructions would not be proposed.
7. `maddedus` is the first Scalar signed/unsigned multiply instruction. The only other signed/unsigned multiply instruction is the specialist `vmsummbm` (bytes only), requires VSX, and is unsuited for big-integer or other general arithmetic.
8. Unresolved: dsld/dsrd are 3-in 3-out (in the Rc=1 variants) where the normal threshold set is 3-in 2-out.
9. Hardware may macro-op fuse inline uses, reducing register use through operand-forwarding and/or higher bit-width ALUs.

Changes

Add the following entries to:

- the Appendices of Book I
- Instructions of Book I added to Section 3.3.9.1
- VA2-Form of Book I Section 1.6.21.1 and 1.6.2

Multiply-Add Extended Double Unsigned

`maddedu RT, RA, RB, RC`

0-5	6-10	11-15	16-20	21-25	26-31	Form
EXT04	RT	RA	RB	RC	XO	VA-Form

Pseudocode:

```
prod[0:127] <- (RA) * (RB)    # Multiply RA and RB, result 128-bit
sum[0:127] <- EXTZ(RC) + prod # Zero extend RC, add product
RT <- sum[64:127]             # Store low half in RT
RS <- sum[0:63]               # RS implicit register, equal to RC
```

Special registers altered:

`None`

The 64-bit operands are (RA), (RB), and (RC). RC is zero-extended (not shifted, not sign-extended). The 128-bit product of the operands (RA) and (RB) is added to (RC). The low-order 64 bits of the 128-bit sum are placed into register RT. The high-order 64 bits of the 128-bit sum are placed into register RS. RS is implicitly defined as the same register as RC.

All three operands and the result are interpreted as unsigned integers.

The differences here to `maddhdu` are that `maddhdu` stores the upper half in RT, where `maddedu` stores the upper half in RS.

The value stored in RT is exactly equivalent to `maddld` despite `maddld` performing sign-extension on RC, because RT is the full mathematical result modulo 2^64 and sign/zero extension from 64 to 128 bits produces identical results modulo 2^64. This is why there is no maddldu instruction.

Programmer's Note: To achieve a big-integer rolling-accumulation effect: assuming the scalar to multiply is in r0, and r3 is used (effectively) as a 64-bit carry, the vector to multiply by starts at r4 and the result vector in r20, instructions may be issued ***`maddedu r20,r4,r0,r3 maddedu r21,r5,r0,r3`*** *etc. where the first* ***`maddedu`*** *will have stored the upper half of the 128-bit multiply into r3, such that it may be picked up by the second* ***`maddedu`****. Repeat inline to construct a larger bigint scalar-vector multiply, as Scalar GPR register file space permits. If register spill is required then r3, as the effective 64-bit carry, continues the chain.*

Examples:

```
# (r0 * r1) + r2, store lower in r4, upper in r2
maddedu r4, r0, r1, r2

# Chaining together for larger bigint (see Programmer's Note above)
# r3 starts with zero (no carry-in)
maddedu r20,r4,r0,r3
maddedu r21,r5,r0,r3
maddedu r22,r6,r0,r3
```

Multiply-Add Extended Double Unsigned/Signed

`maddedus RT, RA, RB, RC`

0-5	6-10	11-15	16-20	21-25	26-31	Form
EXT04	RT	RA	RB	RC	XO	VA-Form

Pseudocode:

```
    if (RB)[0] != 0 then               # workaround no unsigned-signed mul op
        prod[0:127] <- -((RA) * -(RB))
    else
        prod[0:127] <- (RA) * (RB)
    sum[0:127] <- prod + EXTS128((RC))
    RT <- sum[64:127]                  # Store low half in RT
    RS <- sum[0:63]                    # RS implicit register, equal to RC
```

Special registers altered:

`None`

The 64-bit operands are (RA), (RB), and (RC). (RC) is sign-extended to 128-bits and then summed with the 128-bit product of zero-extended (RA) and sign-extended (RB). The low-order 64 bits of the 128-bit sum are placed into register RT. The high-order 64 bits of the 128-bit sum are placed into register RS. RS is implicitly defined as the same register as RC.

Programmer's Note: To achieve a big-integer rolling-accumulation effect: assuming the signed scalar to multiply is in r0, and r3 is used (effectively) as a 64-bit carry, the unsigned vector to multiply by starts at r4 and the signed result vector in r20, instructions may be issued ***`maddedus r20,r4,r0,r3 maddedus r21,r5,r0,r3`*** *etc. where the first* ***`maddedus`*** *will have stored the upper half of the 128-bit multiply into r3, such that it may be picked up by the second* ***`maddedus`****. Repeat inline to construct a larger bigint scalar-vector multiply, as Scalar GPR register file space permits. If register spill is required then r3, as the effective 64-bit carry, continues the chain.*

Examples:

```
# (r0 * r1) + r2, store lower in r4, upper in r2
maddedus r4, r0, r1, r2

# Chaining together for larger bigint (see Programmer's Note above)
# r3 starts with zero (no carry-in)
maddedus r20,r4,r0,r3
maddedus r21,r5,r0,r3
maddedus r22,r6,r0,r3
```

Divide/Modulo Quad-Double Unsigned

`divmod2du RT,RA,RB,RC`

0-5	6-10	11-15	16-20	21-25	26-31	Form
EXT04	RT	RA	RB	RC	XO	VA-Form

Pseudo-code:

```
if ((RA) <u (RB)) & ((RB) != [0]*64) then  # Check RA<RB, for divide-by-0
    dividend[0:127] <- (RA) || (RC)        # Combine RA/RC as 128-bit
    divisor[0:127] <- [0]*64 || (RB)       # Extend RB to 128-bit
    result <- dividend / divisor           # Unsigned Division
    modulo <- dividend % divisor           # Unsigned Modulo
    RT <- result[64:127]                   # Store result in RT
    RS <- modulo[64:127]                   # Modulo in RC, implicit
else                                       # In case of error
    RT <- [1]*64                           # RT all 1's
    RS <- [0]*64                           # RS all 0's
```

Special registers altered:

`None`

The 128-bit dividend is (RA) || (RC). The 64-bit divisor is (RB). If the quotient can be represented in 64 bits, it is placed into register RT. The modulo is placed into register RS. RS is implicitly defined as the same register as RC, similarly to maddedu.

The quotient can be represented in 64-bits when both these conditions are true:

- (RA) < (RB) (unsigned comparison)
- (RB) is NOT 0 (not divide-by-0)

If these conditions are not met, RT is set to all 1's, RS to all 0's.

All operands, quotient, and modulo are interpreted as unsigned integers.

Divide/Modulo Quad-Double Unsigned is a VA-Form instruction that is near-identical to `divdeu` except that:

- the lower 64 bits of the dividend, instead of being zero, contain a register, RC.
- it performs a fused divide and modulo in a single instruction, storing the modulo in an implicit RS (similar to `maddedu`)
- There is no `UNDEFINED` behaviour.

RB, the divisor, remains 64 bit. The instruction is therefore a 128/64 division, producing a (pair) of 64 bit result(s), in the same way that Intel divq works. Overflow conditions are detected in exactly the same fashion as `divdeu`, except that rather than have `UNDEFINED` behaviour, RT is set to all ones and RS set to all zeros on overflow.

Programmer's note: there are no Rc variants of any of these VA-Form instructions. `cmpi` will need to be used to detect overflow conditions: the saving in instruction count is that both RT and RS will have already been set to useful values (all 1s and all zeros respectively) needed as part of implementing Knuth's Algorithm D

For Scalar usage, just as for `maddedu`, `RS=RC` Examples:

```
# ((r0 << 64) + r2) / r1, store in r4
# ((r0 << 64) + r2) % r1, store in r2
divmod2du r4, r0, r1, r2
```

Double-Shift Left Doubleword

`dsld RT,RA,RB,RC`

0-5	6-10	11-15	16-20	21-25	26-30	31	Form
EXT04	RT	RA	RB	RC	XO	Rc	VA2-Form

Pseudo-code:

```
n <- (RB)[58:63]                        # Use lower 6-bits for shift
v <- ROTL64((RA), n)                    # Rotate RA 64-bit left by n bits
mask <- MASK(64, 63-n)                  # 1s mask in MSBs
RT <- (v[0:63] & mask) | ((RC) & ¬mask) # mask-in RC into RT
RS <- v[0:63] & ¬mask                   # part normally lost into RC
overflow = 0                            # Clear overflow flag
if RS != [0]*64:                        # Check if RS is NOT zero
    overflow = 1                        # Set the overflow flag
```

The contents of register RA are shifted left the number of bits specified by (RB) 58:63. The same number of shifted bits are taken from the **right** (LSB) end of register RC and placed into the **rightmost** (LSB) end of the result, RT. Additionally, the MSB (leftmost) bits of register RA that would normally be discarded by a 64-bit left shift are placed into the LSBs of RS.

When Rc=1, the overflow flag in CR0 is set if RS is nonzero, or cleared if it is zero; all other bits of CR0 are set from RT as normal. XER.OV and XER.SO remain unchanged.

Programmer's note: similar to maddedu and divmod2du, dsld can be chained (using RC), effectively using RC as a 64-bit carry-in and carry-out. Arbitrary length Scalar-Vector shift may be performed without the additional masking instructions normally needed.

Special Registers Altered:

```
CR0                     (if Rc=1)
```

Double-Shift Right Doubleword

`dsrd RT,RA,RB,RC`

0-5	6-10	11-15	16-20	21-25	26-30	31	Form
EXT04	RT	RA	RB	RC	XO	Rc	VA2-Form

Pseudo-code:

```
n <- (RB)[58:63]                        # Take lower 6-bits for shift
v <- ROTL64((RA), 64-n)                 # Rotate RA 64-bit left by 64-n bits
mask <- MASK(n, 63)                     # 0's mask, set mask[n:63] to 1'
RT <- (v[0:63] & mask) | ((RC) & ¬mask) #
RS <- v[0:63] & ¬mask
overflow = 0
if RS != [0]*64:
    overflow = 1
```

The contents of register RA are shifted right the number of bits specified by (RB) 58:63. The same number of shifted bits are taken from the **left** (MSB) end of register RC and placed into the **leftmost** (MSB) end of the result, RT. Additionally, the LSB (rightmost) bits of register RA that would normally be discarded by a 64-bit right shift are placed into the MSBs of RS.

When Rc=1, the overflow flag in CR0 is set if RS is nonzero, or cleared if it is zero; all other bits of CR0 are set from RT as normal. XER.OV and XER.SO remain unchanged.

Programmer's note: similar to maddedu and divmod2du, dsrd can be chained (using RC), effectively using RC as a 64-bit carry-in and carry-out. Arbitrary length Scalar-Vector shift may be performed without the additional masking instructions normally needed.

Special Registers Altered:

```
CR0                     (if Rc=1)
```

VA2-Form

Add the following to Book I, 1.6.21.1, VA2-Form

```
    |0      |6     |11     |16     |21  |24|26  |31  |
    | PO    |  RT  |   RA  |   RB  | RC    | XO | Rc |
```

Add 'VA2-Form' to `RA`, `RB`, `RC`, `Rc(31)` `RT` and `XO (26;30)` Fields in Book I, 1.6.2

```
    RA (11:15)
        Field used to specify a GPR to be used as a
        source or as a target.
        Formats: ... VA2, ...

    RB (16:20)
        Field used to specify a GPR to be used as a
        source.
        Formats: ... VA2, ...

    RC (21:25)
        Field used to specify a GPR to be used as a
        source.
        Formats: ... VA2, ...

    Rc (31)
        RECORD bit.
        0    Do not alter the Condition Register.
        1    Set Condition Register Field 0 or Field 1 as
             described in Section 2.3.1, 'Condition Regis-
             ter' on page 30.
        Formats: ... VA2, ...

    RT (6:10)
        Field used to specify a GPR to be used as a target.
        Formats: ... VA2, ...

    XO (26:30)
        Extended opcode field.
        Formats: ... VA2, ...
```

Appendices

```
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Form	Book	Page	Version	mnemonic	Description
VA	I	#	3.2B	maddedu	Multiply-Add Extend Double Unsigned
VA	I	#	3.2B	maddedus	Multiply-Add Extend Double Unsigned Signed
VA	I	#	3.2B	divmod2du	Divide/Modulo Quad-Double Unsigned
VA2	I	#	3.2B	dsld	Double-Shift Left Doubleword
VA2	I	#	3.2B	dsrd	Double-Shift Right Doubleword

[[!tag opf_rfc]]