# Cardinal Processor
# Instruction Set Architecture Manual

This chapter gives an instruction set overview and provides a detailed instruction description. All processor instructions are 32-bits long. Big-Endian byte and bit labeling is used, meaning that bit/byte 0 is the most significant bit. Other conventions are listed in the table below.

| Symbol | Meaning |
|---|---|
| A ← B | Assignment |
| {x, y} | Bit string concatenation |
| {y{x}} | x replicated y times |
| x[y:z] | selection of bits y to z from x |
| x & y | x bitwise ANDed with y |
| x \| y | x bitwise ORed with y |
| x ^ y | x bitwise XORed with y |
| ~x | bitwise inversion of x |
| MEM[EA] | memory contents at effective address EA |
| 0x*value* | Hexadecimal value |
| 0b*value* | Binary Value |
| (rX) | Contents of general purpose register X |
| byte | 8-bit value |
| half-word | 16-bit value |
| word | 32-bit value |
| double-word | 64-bit value |
| INT(x) | Integer value of x |
| x MOD y | x modulo y |

As shown in Figure 1, most Cardinal processor instructions use a three-operand format (R-type) to specify two 64-bit source registers (rA and rB) and one 64-bit destination register (rD). Load and store use a different instruction format (M-type) shown in Figure 2. Note that the above instruction format classifications are generalized, meaning that some instructions may vary from the format described above.

| **6-bits** | **5-bits** | **5-bits** | **5-bits** | **5-bits** | | **6-bits** |
|---|---|---|---|---|---|---|
| 101010 | rD | rA | rB | 000 | WW | |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 23 | 24 25 | 26 31 |

**Figure 1: Format R-type for Arithmetic/Logical Operations**

| **6-bits** | **5-bits** | **5-bits** | **16-bits** |
|---|---|---|---|
| | rD | 00000 | Immediate_Address |
| 0 5 | 6 10 | 11 15 | 16 31 |

**Figure 2: Format M-type for Load/Store Operations**

The control field bits described as follows -

**WW (word width field)**
The 2-bit WW field defines the width of the operands for the R-type instructions. This affects primarily shift, and arithmetic operations. The encoding used for setting operands width is as follows:

| WW value | Operand Width | Assembly Mnemonic |
|---|---|---|
| 00 | 8 | b |
| 01 | 16 | h |
| 10 | 32 | w |
| 11 | 64 | d |

The following table shows the possible subfield indices for the different values of ***WW*** (recall that with Big-Endian labeling used, therefore, subfield 0 is always the most significant regardless of operand size):

| | | Subfield indices within a 64-bit register for different operand widths | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **WW** | **Width** | **MSB** | | | | | | | **LSB** |
| **00** | **8** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **01** | **16** | **0** | | **1** | | **2** | | **3** | |
| **10** | **32** | **0** | | | | **1** | | | |
| **11** | **64** | **0** | | | | | | | |

For multiply and squaring instructions referring to even/odd data-operands, the exact data-operands that participate in the multiply operation are dependent on the operand width specified by the WW field. The above table may be useful for visualizing which data-operands participate based on operand width value. For instance, an "even" data-operands in MULEU instruction for WW = 01 (16-bit data) operations means that only half-words 0 and 2 are used for multiplication.

## Preliminary Encoding of Instruction Set

| Sr. No. | Instruction | Format | Encoding | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 6bits | 5bits | 5bits | 5bits | 5bits | 6bits |
| 1. | **VAND** | R | 101010 | rD | rA | rB | 000WW | 000001 |
| 2. | **VOR** | R | 101010 | rD | rA | rB | 000WW | 000010 |
| 3. | **VXOR** | R | 101010 | rD | rA | rB | 000WW | 000011 |
| 4. | **VNOT** | R | 101010 | rD | rA | 00000 | 000WW | 000100 |
| 5. | **VMOV** | R | 101010 | rD | rA | 00000 | 000WW | 000101 |
| 6. | **VADD** | R | 101010 | rD | rA | rB | 000WW | 000110 |
| 7. | **VSUB** | R | 101010 | rD | rA | rB | 000WW | 000111 |
| 8. | **VMULEU** | R | 101010 | rD | rA | rB | 000WW | 001000 |
| 9. | **VMULOU** | R | 101010 | rD | rA | rB | 000WW | 001001 |
| 10. | **VSLL** | R | 101010 | rD | rA | rB | 000WW | 001010 |
| 11. | **VSRL** | R | 101010 | rD | rA | rB | 000WW | 001011 |
| 12. | **VSRA** | R | 101010 | rD | rA | rB | 000WW | 001100 |
| 13. | **VRTTH** | R | 101010 | rD | rA | **00000** | 000WW | 001101 |
| 14. | **VDIV** | R | 101010 | rD | rA | rB | 000WW | 001110 |
| 15. | **VMOD** | R | 101010 | rD | rA | rB | 000WW | 001111 |
| 16. | **VSQEU** | R | 101010 | rD | rA | 00000 | 000WW | 010000 |
| 17. | **VSQOU** | R | 101010 | rD | rA | 00000 | 000WW | 010001 |
| 18. | **VSQRT** | R | 101010 | rD | rA | 00000 | 000WW | 010010 |
| 19. | **VLD** | M | 100000 | rD | 00000 | Immediate_address | | |
| 20. | **VSD** | M | 100001 | rD | 00000 | Immediate_address | | |
| 21. | **VBEZ** | R | 100010 | rD | 00000 | Immediate_address | | |
| 22. | **VBNEZ** | R | 100011 | rD | 00000 | Immediate_address | | |
| 23. | **VNOP** | R | 111100 | 00000 | 00000 | 00000 | 00000 | 000000 |

The detailed Instruction descriptions for each instruction are as follows:

## vandx – Variable width AND

### vand*w* rD, rA, rB

| | | | | | |
|---|---|---|---|---|---|
| 101010 | rD | rA | rB | 000WW | 000001 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – size) by size
    $rD[i:(i+(size-1))] \leftarrow (rA)[i:(i+(size-1))]\ \&\ (rB)[i:(i+(size-1))]$
```

The 64-bit contents of rA are ANDed with the 64-bit contents of rB, and the result is placed into rD. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The WW field bits do not affect the bit-wise ANDing operation.

## vorx – Variable width OR

### **vor*w*** **rD, rA, rB**

| 101010 | rD | rA | rB | 000WW | 000010 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – size) by size
    $rD[i:(i+(size-1))] \leftarrow (rA)[i:(i+(size-1))] \mid (rB)[i:(i+(size-1))]$
```

The 64-bit contents of rA are ORed with the 64-bit contents of rB, and the result is placed into rD. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The WW field bits do not affect the bit-wise ORing operation.

## vxorx – Variable width XOR

### vxor*w* rD, rA, rB

| | | | | | |
|---|---|---|---|---|---|
| 101010 | rD | rA | rB | 000WW | 000011 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – size) by size
    rD[i:(i+(size – 1))] ← (rA)[i:(i+(size – 1))] ^ (rB)[i:(i+(size – 1))]
```

The 64-bit contents of rA are XOR'ed with the 64-bit contents of rB, and the result is placed into rD. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The WW field bits do not affect the bit-wise XORing operation.

## vnotx – Variable width NOT

### vnot*w* rD, rA

| 101010 | rD | rA | 00000 | 000WW | 000100 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – size) by size
        rD [i: (i+ (size – 1))] ← ~ (rA) [i: (i+ (size – 1))]
```

The 64-bit contents of rA are bit-inverted, and the result is placed into rD. The WW field determines if the 64-bit contents of rA are treated as byte/half-word/word/double-word(s). The WW field bits do not affect the bit-wise inversion operation.

## vmovx – Variable width Move

### vmov*w* rD, rA

| 101010 | rD | rA | 00000 | 000WW | 000101 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to $(64 - size)$ by $size$
    rD $[i : (i + (size - 1))] \leftarrow$ (rA) $[i : (i + (size - 1))]$
```

The entire contents of 64-bit register rA are transferred to destination register rD. The WW field determines if the 64-bit contents of rA are treated as byte/half-word/word/double-word(s). The WW field bits do not affect the bit-wise MOVE operation.

## vaddx – Variable width Add

### vadd*w* rD, rA, rB

| | | | | | |
|---|---|---|---|---|---|
| 101010 | rD | rA | rB | 000WW | 000110 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – $size$) by $size$
    rD[$i:(i+(size-1))$] ← (rA)[$i:(i+(size-1))$] + (rB)[$i:(i+(size-1))$]
```

Each integer byte/half-word/word/double-word of rA is added with the corresponding integer byte/half-word/word/double-word of rB. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The resulting byte/half-word/word/double-word(s) sums are written in the same order into rD. We ignore the generated carry-bit for each integer addition.

## vsubx – Variable width Subtract

### vsub*w* rD, rA, rB

| 101010 | rD | rA | rB | 000WW | 000111 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – size) by size
        rD[$i:(i+(size-1))$] $\leftarrow$ (rA)[$i:(i+(size-1))$] + ~(rB)[$i:(i+(size-1))$] + 1
```

Each integer byte/half-word/word/double-word of rB is subtracted from the corresponding integer byte/half-word/word/double-word of rA. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The resulting integer byte/half-word/word/double-word(s) are written in the same order into rD.
The pseudo-code presented above performs addition of contents of rA with the 2's complement value of corresponding value of rB, which is equivalent to subtracting rB from rA. We ignore any generated overflow bits.

## vmuleux – Variable width Multiply Even Unsigned

### vmuleu*w* rD, rA, rB

| | | | | | |
|---|---|---|---|---|---|
| 101010 | rD | rA | rB | 000WW | 001000 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** | **Output size** |
|---|---|---|
| 00 | 8 | 16 |
| 01 | 16 | 32 |
| 10 | 32 | 64 |

```
For i = 0 to $(64 - 2\times size)$ by $(2 \times size)$
    $rD[i:(i+(2\times size - 1))] \leftarrow (rA)[i:(i+(size - 1))] \times (rB)[i:(i+(size - 1))]$
```

Each even numbered unsigned integer byte/half-word/word of rA is multiplied by the corresponding even numbered unsigned integer byte/half-word/word of rB. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word. The resulting unsigned half-word/word/double-word(s) products are written in the same order into rD. Recall, that the product in the multiplication operation has twice the width of the input operands, therefore, in a 64-bit register, we can only store complete results for up-to 32-bit operands and therefore full 64-bit x 64-bit multiplication is not supported.

## muloux – Variable width Multiply Odd Unsigned

### vmulou*w* rD, rA, rB

| 101010 | rD | rA | rB | 000WW | 001001 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **Input size** | **Output size** |
|---|---|---|
| 00 | 8 | 16 |
| 01 | 16 | 32 |
| 10 | 32 | 64 |

```
For i = 0 to $(64 – 2\times size)$ by $(2 \times size)$
    $rD[i:(i+(2\times size – 1))] \leftarrow (rA)[(i+size):(i+(2\times size – 1))] \times (rB)[(i+size):(i+(2\times size – 1))]$
```

Each odd numbered unsigned integer byte/half-word/word of rA is multiplied by the corresponding odd numbered unsigned integer byte/half-word/word of rB. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word. The resulting unsigned half-word/word/double-word(s) products are written in the same order into rD. Recall, that the product in the multiplication operation has twice the width of the input operands, therefore, in a 64-bit register, we can only store complete results for up-to 32-bit operands and therefore full 64-bit x 64-bit multiplication is not supported.

## <u>vsllx – Variable width Shift Left Logical</u>

### vsll*w* rD, rA, rB

| | | | | | |
|---|---|---|---|---|---|
| 101010 | rD | rA | rB | 000WW | 001010 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** | **Bits** |
|---|---|---|
| 00 | 8 | 3 |
| 01 | 16 | 4 |
| 10 | 32 | 5 |
| 11 | 64 | 6 |

```
For i = 0 to (64 – size) by size
    $s \leftarrow (rB)[(i+ size – bits):(i + size – 1)]$
    $rD[i:(i+(size – 1))] \leftarrow \{(rA)[(i+ s):(i+ size – 1], s\{0\}\}$
```

Each byte/half-word/word/double-word(s) of rA are shifted left by the value given by the number of bits specified by the lower significant bits (LSB) of the corresponding data-fields contained as contents of rB, and inserting zeros into the least significant bits (LSB) of each data-field of the result. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s), and it also determines how many lower significant bits are to be considered for calculation of shift-amount. The resulting byte/half-word/word/double-word(s) are written in the same order into rD.

## vsrlx – Variable width Shift Right Logical

### vsrl*w* rD, rA, rB

| 101010 | rD | rA | rB | 000WW | 001011 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** | **Bits** |
|---|---|---|
| 00 | 8 | 3 |
| 01 | 16 | 4 |
| 10 | 32 | 5 |
| 11 | 64 | 6 |

```
For i = 0 to (64 – size) by size
    $s \leftarrow (rB)[(i + size - bits) : (i + size - 1)]$
    $rD[i : (i + (size - 1))] \leftarrow \{s\{0\}, (rA)[i : (i + size - s - 1)]\}$
```

Each byte/half-word/word/double-word(s) of rA are shifted right by the value given by the number of bits specified by the lower significant bits (LSB) of the corresponding data-fields contained as contents of rB, and inserting zeros into the most significant bits (MSB) of each data-field of the result. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s), and it also determines how many lower significant bits are to be considered for calculation of shift-amount. The resulting byte/half-word/word/double-word(s) are written in the same order into rD.

## vsrax – Variable width Shift Right Arithmetic

### vsra*w* rD, rA, rB

| 101010 | rD | rA | rB | 000WW | 001100 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| WW value | size | Bits |
|---|---|---|
| 00 | 8 | 3 |
| 01 | 16 | 4 |
| 10 | 32 | 5 |
| 11 | 64 | 6 |

```
For i = 0 to (64 – size) by size
    s ← (rB) [(i+ size –bits) :(i+size – 1)]
    rD[i:(i+(size – 1))] ← {s{(rA)[i]}, (rA)[i :(i + size – s – 1]}
```

Each byte/half-word/word/double-word(s) of rA are shifted right by the value given by the number of bits specified by the lower significant bits (LSB) of the corresponding data-fields contained as contents of rB, and sign extending the most significant bit into the most significant bits (MSB) of each data-field of the result. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s), and it also determines how many lower significant bits are to be considered for calculation of shift-amount. The resulting byte/half-word/word/double-word(s) are written in the same order into rD.

## vrtthx – Variable width Rotate by Half

### vrtthw rD, rA

| 101010 | rD | rA | 00000 | 000WW | 001101 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

```
For i = 0 to (64 – size) by size
    rD[i:(i+(size – 1))] ← {(rA)[i+ size/2:(i+(size – 1))], (rA)[i:(i+(size/2 – 1))]}
```

Each byte/half-word/word/double-word(s) of rA are rotated right by half of the size as specified by WW field bits. The result is that lower nibble/byte/half-word/word is swapped with the higher nibble/byte/half-word/word respectively. The WW field determines if the 64-bit contents of rA are treated as byte/half-word/word/double-word(s), and it also determines the size of the rotation. The resulting byte/half-word/word/double-word(s) are written in the same order into rD.

## vdivux – Variable width Division Integer Unsigned

### vdivu*w* rD, rA, rB

| 101010 | rD | rA | rB | 000WW | 001110 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **Input size** | **Output size** |
|---|---|---|
| 00 | 8 | 8 |
| 01 | 16 | 16 |
| 10 | 32 | 32 |
| 11 | 64 | 64 |

For i = 0 to (64 – *size*) by *size*

**rD[*i:(i+(size – 1))*] ← INT ( (rA) [*i:(i+(size – 1))*] / (rB) [*i:(i+(size – 1))*] )**

Each unsigned integer byte/half-word/word/double-word of rA is divided by the corresponding unsigned integer byte/half-word/word/double-word of rB. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The resulting integer results of the division is written in the same order into rD.

## vmodux – Variable width Modulo Integer Unsigned

### vmodu*w* rD, rA, rB

| 101010 | rD | rA | rB | 000WW | 001111 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **Input size** | **Output size** |
|---|---|---|
| 00 | 8 | 8 |
| 01 | 16 | 16 |
| 10 | 32 | 32 |
| 11 | 64 | 64 |

For i = 0 to (64 – *size*) by *size*
rD[*i:(i+(size – 1))*] ← INT( (rA)[*i:(i+(size – 1))*] MOD (rB)[*i:(i+(size – 1))*] )

Each unsigned integer byte/half-word/word/double-word of rA is divided by the corresponding unsigned integer byte/half-word/word/double-word of rB. The WW field determines if the 64-bit contents of rA and rB are treated as byte/half-word/word/double-word(s). The resulting integer remainder of the division is written in the same order into rD.

## vsqeux – Variable width Square Even Unsigned

### vsqeu*w* rD, rA

| 101010 | rD | rA | 00000 | 000WW | 010000 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** | **Output size** |
|---|---|---|
| 00 | 8 | 16 |
| 01 | 16 | 32 |
| 10 | 32 | 64 |

For i = 0 to $(64 - 2\times size)$ by $(2 \times size)$

$$rD\,[i{:}\,(i + (2\times size - 1))] \leftarrow \{\,(rA)[i{:}\,(i + (size - 1))]\,\}^2$$

Each even numbered unsigned integer byte/half-word/word of rA is squared. The WW field determines if the 64-bit contents of rA are treated as byte/half-word/word. The resulting unsigned half-word/word/double-word(s) products are written in the same order into rD. Recall, that the product of the squaring operation has twice the width of the input operands, therefore, in a 64-bit register, we can only store complete results for up-to 32-bit operands and therefore full 64-bit x 64-bit multiplication is not supported.

## vsqoux – Variable width Square Odd Unsigned

### vsqou*w* rD, rA

| 101010 | rD | rA | 00000 | 000WW | 010001 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **Input size** | **Output size** |
|---|---|---|
| 00 | 8 | 16 |
| 01 | 16 | 32 |
| 10 | 32 | 64 |

For i = 0 to ($64 - 2\times size$) by ($2 \times size$)

rD [$i$: ($i$+ ($2\times size - 1$))] $\leftarrow$ { (rA)$[(i + size) : (i + (2 \times size - 1))]$ } $^2$

Each odd numbered unsigned integer byte/half-word/word of rA is squared. The WW field determines if the 64-bit contents of rA are treated as byte/half-word/word. The resulting unsigned half-word/word/double-word(s) products are written in the same order into rD. Recall, that the product in the squaring operation has twice the width of the input operands, therefore, in a 64-bit register, we can only store complete results for up-to 32-bit operands and therefore full 64-bit x 64-bit multiplication is not supported.

## vsqrtux – Variable width Square root Integer Unsigned

### vbsqrtu*w* *rD, rA*

| 101010 | rD | rA | 00000 | 000WW | 010010 |
|---|---|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| WW value | size |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

For i = 0 to (64 – *size*) by *size*

rD [$i$: ($i$+ (*size* – 1))] ← $\sqrt{rA\ [i : (i + size - 1)]}$

This instruction computes the square-root of the unsigned integer byte/half-word/word/double-word of rA. The WW field determines if the 64-bit contents of rA are treated as byte/half-word/word/double-word(s). The resulting unsigned integer square root value(s) are written in the same order into the rD.

## vldx – Load Register from data memory

### vld*w* rD, Immediate_address

| 100000 | rD | 00000 | Immediate_address |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

EA ← {16{0}, immediate_address}
rD ← MEM [EA]

The immediate address is assumed to be in terms of 64-bit words. The data-memory provided is only accessible in terms of 64-bit words. The 64-bit value at the memory location specified by EA is then loaded into rD.

## <u>vsdx – Store Register to data memory</u>

### vsd*w* rD, Immediate_address

| 100001 | rD | 00000 | Immediate_address |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

EA ← {16{0}, immediate_address}
MEM [EA] ← (**rD**)

The immediate address is assumed to be in terms of 64-bit wide words. The data-memory provided is only accessible in terms of 64-bit words. The 64-bit contents specified by the register rD are written to the data-memory location specified in the instruction.

## vbeqx – Branch if Equal to Zero

### vbeq*w* rD, Immediate_address

| 100010 | rD | 00000 | Immediate_address |
|---|---|---|---|
| 0 5 | 6 10 | 11 15 | 16 31 |

If (rD) == 0
PC ← immediate_address
($0 \leq$ Immediate_address $\leq (2^{16} - 1)$)

If the contents of register rD are zero, then the branch is executed. The branch target address is the 16-bit immediate address as specified in the instruction. In this project, we are implementing absolute branch addressing scheme, therefore, with a branch-target address of 16-bits, we can jump between ($0 \leq$ Immediate_address $\leq (2^{16} - 1)$) address space.

## vbneqx – Branch if Not Equal to Zero

### vbneq*w* rD, Immediate_address

| | | | |
|---|---|---|---|
| 100011 | rD | 00000 | Immediate_address |
| 0 5 | 6 10 | 11 15 | 16 31 |

If (rD) != 0
    PC ← Immediate_address
    (0 ≤ Immediate_address ≤ ($2^{16}$ - 1))

If the contents of register rD are not zero, then the branch is executed. The branch target address is the 16-bit immediate address as specified in the instruction. In this project, we are implementing absolute branch addressing scheme, therefore, with a branch-target address of 16-bits, we can jump between (0 ≤ Immediate_address ≤ ($2^{16}$ - 1)) address space.

## vnopx – No Operation

### vbnop*w*

| | | | | | |
|---|---|---|---|---|---|
| 111100 | 00000 | 00000 | 00000 | 00000 | 000000 |
| 0 5 | 6 10 | 11 15 | 16 20 | 21 25 | 26 31 |

Variable values in the following equations are as follows:

| **WW value** | **size** |
|---|---|
| 00 | 8 |
| 01 | 16 |
| 10 | 32 |
| 11 | 64 |

This instruction is equivalent to inserting bubble in the design. It performs no useful task. When this instruction is executed, it should not read/write to memory location and should not update register-file contents.