# *Locutus CP-1600X Instruction Set Extensions V1.0*

*3-Dec-2019, A*

Copyright © 2019 — Joseph Zbiciak — Left Turn Only


ONLY
ONE WAY

# Background

The Locutus CP-1600X Instruction Set transparently extends the base instruction set of the General Instrument CP-1600 and CP-1610 processors for programs running on a Locutus cartridge. A future revision of JLP may also incorporate these instructions. The instruction set is also implemented in jzIntv.

Previously, JLP and jzIntv provided memory-mapped acceleration for multiplies, divides, CRCs, and random number generation. The CP-1600X represents a more tightly integrated approach, leveraging various unique aspects of the CP-1600 instruction set and implementation. The resulting extensions look, feel, and behave more like instructions, and less like memory-mapped accelerators.

The CP-1600X extensions were inspired in part by the phantom, vaporware CP-1620 coprocessor, documented in various early CP-1600 and GIMINI manuals. It was intended to extend the CP-1600 instruction set, similar to how an 8087 extends an 8086. As far as I can tell, this chip never materialized.

# Design Goals

## High Level Goals

- 100% compatibility with existing CP-1600 machine code.
    - Provided that bits 15:10 of the primary opcode word are zero, as recommended by GI, and as implemented in most (all?) assemblers currently in use.
- Seamless extension of existing instructions, such as:
    - Enhanced addressing modes.
    - Extended register set.
- Similar cost to native instructions, both in size and speed.
    - A 16×16 multiply on JLP requires two MVOs and at least one MVI.
        - That takes a minimum of 26 cycles, and more likely 32 cycles.
        - That costs 3 to 6 instruction words.
        - ~10 cycles and 1-2 instruction words is a more reasonable target.
    - JLP instructions suffer interrupt atomicity issues.
        - This can be solved for 16-bit results, but 32-bit results require masking interrupts or constraining how the instructions get used:
          e.g. MVO; MVO; MVI is safe. MVO; MVO; MVI; MVI is not. Why? MVO is non-interruptible[1] but MVI is.
- Fits with overall feel of CP-1600 while providing useful functions for video games.

[1] Non-interruptible means no interrupt can be taken between that instruction and the one that follows it.

## Constraints / Anti-Goals

- Must not require internal modifications to the original unit. All operations must be available via a hardware cartridge plugged into a standard Intellivision cartridge port.
- Should not require *rewriting* any of the bus phase signals—e.g. generating different BC1_OUT/BC2_OUT/BDIR_OUT as compared to BC1_IN/BC2_IN/BDIR_IN—to maximize compatibility across Intellivision versions and Intellivision peripherals.
- Must not be *over the top*: It's possible to implement a program entirely in an external machine, using the Intellivision solely for access to the controllers and display. That is explicitly an *anti-goal*. These instructions should feel like coprocessor extensions that would be reasonable circa 1984.
- It's OK if the additional instructions are only available to code that executes from Locutus or other specialized cartridges. That is, code running from Intellivision RAM does not have access to the extended instructions.

# Technical Background

## CP-1600 Instruction Set Format Summary

| Format | | | Words | Description |
|---|---|---|---|---|
| `0 000 000 0oo` | | | 1 | Implied 1-op instructions |
| `0 000 000 100` | `bb pppppp ii` | `pppppppppp` | 3 | Jump instructions |
| `0 000 000 1oo` | | | 1 | Implied 1-op instructions |
| `0 000 ooo ddd` | | | 1 | 1-op src/dst instructions |
| `0 000 110 0dd` | | | 1 | GSWD |
| `0 000 110 1om` | | | 1 | NOP, SIN |
| `0 001 ooo mrr` | | | 1 | Rotate / shift instructions |
| `0 ooo sss ddd` | | | 1 | 2-op arithmetic, reg-to-reg |
| `1 000 zxc ccc` | 16-bit offset | | 2 | Branch instructions |
| `1 ooo 000 ddd` | 16-bit address | | 2 | 2-op arithmetic, direct mode |
| `1 ooo mmm ddd` | | | 1 | 2-op arithmetic, indirect mode |
| `1 ooo 111 ddd` | 16-bit immediate | | 2[2] | 2-op arithmetic, immediate mode |

[2] 2 words for the immediate value (3 words total) if instruction is preceded by SDBD, with the immediate in bits 0..7 of each word.

## Opcode field key

| | |
|---|---|
| oo | Opcode field (meaning depends on format) |
| sss | Source register (R0 to R7) |
| ddd | Destination register (R0 to R7) |
| 0dd | Destination register (R0 to R3) |
| cccc | Branch condition code (table below) |
| x | External branch condition (0 = internal, 1 = external) |
| z | Branch displacement direction (1 = negative) |
| m | Shift amount (0 = shift by 1; 1 = shift by 2) |
| bb | Branch return register (00 = R4, 01 = R5, 10 = R6, 11 = None) |
| ii | Branch interruptibility flag (00 = No Change, 01 = EIS, 10 = DIS, 11 = Reserved) |

## Conditional branch codes

| | n = 0 | n = 1 |
|---|---|---|
| n000 | Always | Never |
| n001 | Carry set / Unsigned Greater or Equal | Carry clear / Unsigned Less Than |
| n010 | Overflow set | Overflow clear |
| n011 | Positive (S = 0) | Negative (S = 1) |
| n100 | Equal (Z = 1) | Not equal (Z = 0) |
| n101 | Signed less than | Signed greater than or equal |
| n110 | Signed less than or equal | Signed greater than |
| n111 | Unequal sign and carry (S ≠ C) | Equal sign and carry (S = C) |

# CP-1600 Instruction Extension Opportunities

## 10 Bit Opcodes

The CP-1600 ignores bits 10 through 15 of the first word of every instruction. The canonical encoding for each CP-1600 instruction places zeros in these 6 bits. Thus, if these bits are non-zero, the CPU will execute the instruction *as if* the upper bits are 0. External hardware can interpret those bits, however, and use that information to curate the CPU's view of the system.

## Direct Addressing

The CP-1600 bus protocol supports a rather unique bus phase: ADAR, Addressed Data to Address Register. Instructions that use direct addressing use the ADAR bus phase to access the corresponding operand.

Direct addressing instructions look similar to indirect-addressing instructions, except that they specify R0 as the indirect address register. The CP-1600 knows to interpret this as a direct-address instruction. The resulting access pattern differs as follows for instructions that read, and do not use SDBD.

| | **Indirect Mode** | | **Direct Mode** | |
|---|---|---|---|---|
| **Cycle** | **Bus Phase** | **Description** | **Bus Phase** | **Description** |
| 1 | BAR | Address of instruction | BAR | Address of instruction |
| 2 | NACT | No action | NACT | No action |
| 3 | DTB | Fetch instruction | DTB | Fetch instruction |
| 4 | NACT | No action | NACT | No action |
| 5 | BAR | Address of operand from register | BAR | Address of operand following instruction |
| 6 | NACT | No action | NACT | No action |
| 7 | DTB | Read indirect operand | ADAR | Read address of operand; latch as next address similar to BAR |
| 8 | - | | NACT | No action |
| 9 | - | | DTB | Read direct operand |

The MVO instruction proceeds similarly to other instructions, replacing the DTB bus phase with two consecutive bus phases: DW and DWS.

In any case, external hardware has the opportunity to modify the address presented during ADAR before the CPU and the system acts on it. And, for direct-mode instructions that target R7, external hardware can further synthesize interesting branch constructs.

And, more generally, even without ISA extensions, direct addressing allows external memory locations to look a lot like registers, at least for the first operand of most instructions and the second operand of a MVO instruction.

## MVOI - MoVe Out Immediate

If you look carefully at the instruction encodings, it becomes apparent that the CP-1600 does not *actually* implement an immediate mode for its core opcodes. Rather, it implements "indirect via the program counter." For most instructions, the difference is meaningless: The CPU fetches the immediate operand via R7, incrementing it as necessary.

For MVO, however, it's a bit different. `MVO@ …, R7` stores the contents of a register at the address following an instruction. The General Instrument documentation calls the resulting instruction MVOI: "MoVe Out, Immediate." For programs stored in ROM, this instruction is essentially useless. For programs stored in RAM, it *might* be useful; however, in practice, its usefulness is limited.

For Locutus' CP-1600X, MVOI presents a great opportunity:

- It writes the current value of a register onto the bus.
- The CPU ignores the upper 6 bits of the opcode for the MVOI.
- The CPU ignores the contents of the 16-bit word that follows the MVOI.
- Because it is a MVO instruction, it is also uninterruptible.
- Writes provide two bus phases for write data—DW and DWS—which potentially translates into more time for computation.

Ultimately, this provides a great expansion mechanism with the following properties:

- 22 bits of expansion opcode.
- Fast access to one of the CP-1600's internal registers as an argument.
- An additional bus cycle for computation.
- Non-interruptible, enabling support for certain atomic operations.

# CP-1600X Instruction Set Extension Summary

The Locutus CP-1600X Instruction Set provides the following extensions:

- Extended registers X0 through XF and PV.
- Extended addressing modes for the core CP-1600 instruction set: MVO, MVI, ADD, SUB, CMP, AND, and XOR.
- Atomic operation support.
- Extended register-to-register instructions.
- Specialized branching and looping instructions.

## Extended Register Set

The CP-1600X provides 17 additional registers. These registers break down into three groups:

- X0 through X7. These registers are available as address registers for extended addressing modes, as well as extended register-to-register operations.
- X8 through XF. These registers are available to extended register-to-register operations.
- PV. This special register holds the *previous value* associated with certain extended operations. It provides a key component of CP-1600X's atomic operation support.

CP-1600X maps these registers into the CP-1600 address space. That allows direct-mode CP-1600 instructions to treat the additional registers similarly to native registers, subject to the restrictions of direct addressing operands.

| Register | Address | Register | Address |
|---|---|---|---|
| X0 | $9F90 | X8 | $9F98 |
| X1 | $9F91 | X9 | $9F99 |
| X2 | $9F92 | XA | $9F9A |
| X3 | $9F93 | XB | $9F9B |
| X4 | $9F94 | XC | $9F9C |
| X5 | $9F95 | XD | $9F9D |
| X6 | $9F96 | XE | $9F9E |
| X7 | $9F97 | XF | $9F9F |
| | | PV | $9F8D |

## Register Pairs

Some instructions use a 32-bit register pair, referred to here in the instruction descriptions as src_hi:src_lo or dst_hi:dst_lo.

In the instruction itself, specify the lower numbered register of the pair. This becomes the 'lo' half of the pair. The next higher numbered register becomes 'hi' half. For XF, the next higher numbered register is X0.

## The PV Register

The PV register captures the *previous* contents of an extended register *or* memory location under the following circumstances:

- An indirect write to Locutus memory (RAM, ROM, or WOM) with MV0@, *excluding* extended addressing modes.
- An *atomic instruction* that writes to Locutus memory (RAM, ROM, or WOM).
- An *extended register-to-register instruction* that writes to X0 through XF.

In each case, CP-1600X captures the previous value of the target location for the write into PV. For extended register-to-register instructions that write to a pair of registers, PV captures dst_lo.

# Extended Addressing Modes

CP-1600X provides extended addressing modes to the core 7 instructions: MVO, MVI, ADD, SUB, CMP, AND, and XOR. In addition to that, the extended address encoding enables four atomic operation instructions: MVO (which behaves as an atomic exchange), ATADD, ATAND, and ATOR.

## Extended Addressing Modes

The default macro package provides spellings for CP-1600X's new addressing modes, in addition to alternate spellings for CP-1600's original addressing modes. The tables below list the full complement of address modes and their spelling.

### Opcode Encoding

The extended addressing modes build on CP-1600's direct addressing mode. These instructions require two words, and fit the following encoding:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | amode | | extreg | | | 1 | opcode | | | 0 | 0 | 0 | reg | | |

| offset |
|---|

#### Opcode Field Definitions

| Field | Meaning |
|---|---|
| amode | Addressing mode (see below). |
| extreg | External register X0 through X7. (000b = X0, 111b = X7, etc.) |
| opcode | CP-1600 instruction opcode |
| reg | Internal register R0 through R7. (000b = R0, 111b = R7, etc.) |
| offset | Address offset to apply in extended addressing modes (amode ≠ 0 or extreg ≠ 0) |

#### Assembler Aliases for Native Modes: amode = 00b, extreg = 000b

| Syntax | Meaning | Syntax | Meaning |
|---|---|---|---|
| @R1 | Indirect through R1. | @--R6 | Indirect through R6, pre-decrement. |
| @R2 | Indirect through R2. | @--SP | Alias for @--R6. |
| @R3 | Indirect through R3. | @R6++ | Indirect through R6, post-increment. |
| @R4++ | Indirect through R4, post-increment. | @SP++ | Alias for @R6++. |
| @R5++ | Indirect through R5, post-increment. | @R7++ | Indirect through R7, post-increment. |
| | | @PC++ | Alias for @R7++. |

### Effective Address Mode: amode = 00b, extreg ≠ 000b, opcode ≠ MVO

These modes do not access memory; rather, they return the address computed as an immediate operand, similar to the LEA instructions on 8086 and 68000.

| Syntax | Meaning | Syntax | Meaning |
|---|---|---|---|
| - | *X0 not supported in this mode.* | &X4(ofs) | Return X4 + *ofs* as an immediate. |
| &X1(ofs) | Return X1 + *ofs* as an immediate. | &X5(ofs) | Return X5 + *ofs* as an immediate. |
| &X2(ofs) | Return X2 + *ofs* as an immediate. | &X6(ofs) | Return X6 + *ofs* as an immediate. |
| &X3(ofs) | Return X3 + *ofs* as an immediate. | &X7(ofs) | Return X7 + *ofs* as an immediate. |

### Three Operand Add: amode = 00b, extreg ≠ 000b, opcode = MVO

The effective address mode doesn't make much sense for MVO, as there's no way to see the generated effective address. Therefore, CP-1600X repurposes this encoding to implement a three operand add:

```
   ADD3x  srcreg, immediate, extreg
```

This allows adding a 16-bit immediate value to a CPU register, storing the result in an extension register.

*Restrictions*:

- *srcreg* must be one of R0 through R7
- *extreg* must be one of X1 through X7

### Special Operations: amode = 00b, extreg ≠ 000b, reg = 111b

The CP-1600X provides a family of Extended Conditional Branches. These are encoded as amode=00b instructions that target R7. The following table provides a summary.

| Opcode | Original Mnemonic | Extended ISA Mnemonic | Description |
|---|---|---|---|
| 001 | MVO | - | No change / reserved. |
| 010 | MVI | TSTBNZ | Test Xreg and branch if non-zero. |
| 011 | ADD | TXSER / RXSER | Specialized serial transmit/receive with branch. |
| 100 | SUB | - | No change / reserved. |
| 101 | CMP | - | No change / reserved. |
| 110 | AND | - | No change / reserved. |
| 111 | XOR | DECBNZ | Decrement Xreg and branch if non-zero. |

Indirect-Indexed Modes: amode = 01b

| Syntax | Meaning | Syntax | Meaning |
|---|---|---|---|
| @X0(ofs) | Access location (X0 + *ofs*) | @X4(ofs) | Access location (X4 + *ofs*) |
| @X1(ofs) | Access location (X1 + *ofs*) | @X5(ofs) | Access location (X5 + *ofs*) |
| @X2(ofs) | Access location (X2 + *ofs*) | @X6(ofs) | Access location (X6 + *ofs*) |
| @X3(ofs) | Access location (X3 + *ofs*) | @X7(ofs) | Access location (X7 + *ofs*) |

Indirect Post-Increment, Post-Decrement: amode = 10b

| Syntax | Meaning | Syntax | Meaning |
|---|---|---|---|
| @X0++(ofs) | Access location (X0);<br>X0 ⟸ X0 + *ofs* | @X0--(ofs) | Access location (X0);<br>X0 ⟸ X0 - *ofs* |
| @X1++(ofs) | Access location (X1);<br>X1 ⟸ X1 + *ofs* | @X1--(ofs) | Access location (X1);<br>X1 ⟸ X1 - *ofs* |
| @X2++(ofs) | Access location (X2);<br>X2 ⟸ X2 + *ofs* | @X2--(ofs) | Access location (X2);<br>X2 ⟸ X2 - *ofs* |
| @X3++(ofs) | Access location (X3);<br>X3 ⟸ X3 + *ofs* | @X3--(ofs) | Access location (X3);<br>X3 ⟸ X3 - *ofs* |
| @X4++(ofs) | Access location (X4);<br>X4 ⟸ X4 + *ofs* | @X4--(ofs) | Access location (X4);<br>X4 ⟸ X4 - *ofs* |
| @X5++(ofs) | Access location (X5);<br>X5 ⟸ X5 + *ofs* | @X5--(ofs) | Access location (X5);<br>X5 ⟸ X5 - *ofs* |
| @X6++(ofs) | Access location (X6);<br>X6 ⟸ X6 + *ofs* | @X6--(ofs) | Access location (X6);<br>X6 ⟸ X6 - *ofs* |
| @X7++(ofs) | Access location (X7);<br>X7 ⟸ X7 + *ofs* | @X7--(ofs) | Access location (X7);<br>X7 ⟸ X7 - *ofs* |

### Indirect Pre-Increment, Pre-Decrement: amode = 11b

| Syntax | Meaning | Syntax | Meaning |
|---|---|---|---|
| `@++X0(ofs)` | Access location (X0 + *ofs*);<br>X0 ⟸ X0 + *ofs* | `@--X0(ofs)` | Access location (X0 - *ofs*);<br>X0 ⟸ X0 - *ofs* |
| `@++X1(ofs)` | Access location (X1 + *ofs*);<br>X1 ⟸ X1 + *ofs* | `@--X1(ofs)` | Access location (X1 - *ofs*);<br>X1 ⟸ X1 - *ofs* |
| `@++X2(ofs)` | Access location (X2 + *ofs*);<br>X2 ⟸ X2 + *ofs* | `@--X2(ofs)` | Access location (X2 - *ofs*);<br>X2 ⟸ X2 - *ofs* |
| `@++X3(ofs)` | Access location (X3 + *ofs*);<br>X3 ⟸ X3 + *ofs* | `@--X3(ofs)` | Access location (X3 - *ofs*);<br>X3 ⟸ X3 - *ofs* |
| `@++X4(ofs)` | Access location (X4 + *ofs*);<br>X4 ⟸ X4 + *ofs* | `@--X4(ofs)` | Access location (X4 - *ofs*);<br>X4 ⟸ X4 - *ofs* |
| `@++X5(ofs)` | Access location (X5 + *ofs*);<br>X5 ⟸ X5 + *ofs* | `@--X5(ofs)` | Access location (X5 - *ofs*);<br>X5 ⟸ X5 - *ofs* |
| `@++X6(ofs)` | Access location (X6 + *ofs*);<br>X6 ⟸ X6 + *ofs* | `@--X6(ofs)` | Access location (X6 - *ofs*);<br>X6 ⟸ X6 - *ofs* |
| `@++X7(ofs)` | Access location (X7 + *ofs*);<br>X7 ⟸ X7 + *ofs* | `@--X7(ofs)` | Access location (X7 - *ofs*);<br>X7 ⟸ X7 - *ofs* |

### Example Encodings For Extended Addresses

Recall the opcode format:

| 15 | 14 13 | 12 11 10 | 9 | 8 7 6 | 5 | 4 | 3 | 2 1 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | amode | extreg | 1 | opcode | 0 | 0 | 0 | reg |
| offset | | | | | | | | |

Here are some example encodings. Note that offset field is in decimal unless stated otherwise; other fields are in binary.

| Instruction | **0** | amode | extreg | **1** | opcode | **000** | reg | offset |
|---|---|---|---|---|---|---|---|---|
| `MVI @X3++(1), R5` | 0 | 10 | 011 | 1 | 010 | 000 | 101 | 1 |
| `ADD3x R2, 42, X7` | 0 | 00 | 111 | 1 | 001 | 000 | 111 | 42 |
| `SUB &X6(123), R1` | 0 | 00 | 110 | 1 | 100 | 000 | 001 | 123 |
| `MVO R3, @--X2(5)` | 0 | 11 | 010 | 1 | 001 | 000 | 011 | -5 |
| `MVO R4, @X2(4)` | 0 | 01 | 010 | 1 | 001 | 000 | 100 | 4 |

# Atomic Instructions

The atomic instructions modify the behavior of MVO for *indirect mode* instructions. The atomic instruction updates a value in memory atomically, and returns the previous value in the PV register.

## Opcode Encoding

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| atomic opcode | | | | | | 1 | MVO (001) | | | dstreg[3] | | | srcreg | | |

| Atomic Opcode | Mnemonic | Operation |
|---|---|---|
| 000000 | MVO | PV ⟸ @dstreg; @dstreg ⟸ srcreg |
| 000001 | ATADD | PV ⟸ @dstreg; @dstreg ⟸ @dstreg + srcreg |
| 000010 | ATAND | PV ⟸ @dstreg; @dstreg ⟸ @dstreg & srcreg |
| 000011 | ATOR | PV ⟸ @dstreg; @dstreg ⟸ @dstreg \| srcreg |
| 0001xx - 1xxxxx | - | *Reserved.* |

## Operation

These instructions behave similarly to MVO, in that the CPU writes to memory, and it treats the instruction as non-interruptible. CP-1600X captures the previous value in memory (provided the write targets memory controlled by Locutus) in the PV register.

Therefore, a complete atomic operation consists of an ATxxx instruction immediately followed by a MVI instruction that captures PV into an internal register.

### Atomic Add Example

```
ATADD R1, @R2 ; Add value in R1 to value @R2; record previous value @R2 in PV.
MVI   PV, R3  ; Capture the previous value @R2 into R3.
```

[3] Note: dstreg ≠ 000b.

# Extended Register-to-Register Instructions

The CP-1600X provides a number of three-operand instructions that operate register-to-register. Most of these instructions produce results in extended registers, but take arguments from both CPU and extended registers. Because these instructions build on `MVOI`, these instructions are also non-interruptible.

## Numeric Formats

Many of the compute instructions are available in multiple flavors:

- Signed integer
- Unsigned integer
- Signed fixed point
- Unsigned fixed point
- Binary Coded Decimal (BCD)

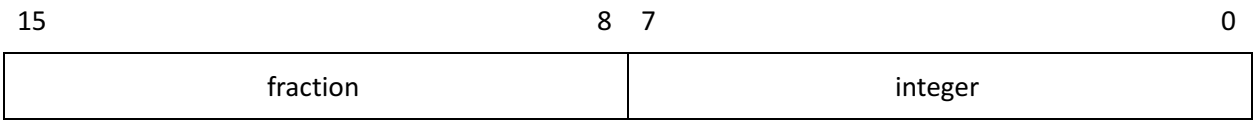
### Integer Formats

The signed and unsigned integer formats are fairly straightforward 2s complement formats.

Signed 16-bit integers have the range -32768 to 32767 (-0x8000 to 0x7FFF), and unsigned 16-bit integers have the range 0 to 65535 (0x0000 to 0xFFFF). Signed and unsigned 32-bit integer formats are similar, just extended to 32 bits.

### Fixed Point Formats

The signed and unsigned fixed point formats are a little more interesting. Both fixed point formats are 16 bits, providing 8 fraction bits and 8 integer bits. The fixed-point formats are *also* rotated by 8 bits, so that the integer portion is in bits 0 .. 7 and the fraction portion is in bits 8 .. 15. This corresponds to IntyBASIC's fixed point data type.

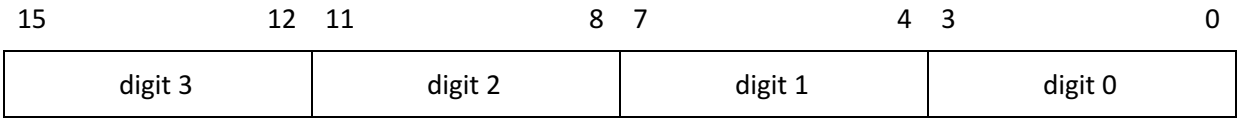



For example, the number 123.45, which is approximated as $7B.73_{16}$, would be stored as `0x737B` in CP-1600X's fixed point format.

Fixed point instructions generally have an FX in their mnemonic.

### BCD Formats

CP-1600X also supports both 16-bit and 32-bit unsigned BCD format, as well as the capability to extend to multiples of 16 bits. The 16-bit BCD format divides a 16-bit word into four 4-bit fields, each holding a single decimal digit. The 32-bit BCD format extends this to a second 16-bit word.




Traditional BCD formats typically assign bit patterns $0000_2$ to $1001_2$ to digits 0 through 9, and leave $1010_2$ through $1111_2$ as undefined.

CP-1600X, however, takes a different stance: Encodings $0000_2$ through $1001_2$ form canonical digits, and each BCD computation will leave its digits in canonical form. However, on input, the instructions consistently treat encodings $1010_2$ through $1111_2$ as having the values 10 through 15.

Thus, the hexadecimal value $FFFF_{16}$ behaves numerically as $15 \cdot 1000 + 15 \cdot 100 + 15 \cdot 10 + 15 = 16665$.

To support extended precision arithmetic, some BCD instructions produce a carry or borrow output that can be consumed by other BCD instructions. Due to the nature of CP-1600X's extended BCD range, this carry/borrow output is a signed 3-bit number:

| Encoding | Value |
|---|---|
| 000 | +0 |
| 001 | +1 |
| 010 | +2 |
| 011 | +3 |

| Encoding | Value |
|---|---|
| 100 | -4 |
| 101 | -3 |
| 110 | -2 |
| 111 | -1 |

In practice, CP-1600X only generates carry/borrow in the range [-2, +3]. The compute instructions should perform as expected if you provide the full range as inputs. Instructions that consume BCD carry/borrow information from a register only examine the 3 LSBs.

## Opcode Formats

These instructions build upon the base `MVOI` opcode. CP-1600X looks at bits 0 - 2 and 10 - 15 of the `MVOI` instruction itself, as well as all 16 bits of the word following the `MVOI` opcode. This creates a 25-bit opcode space.

## Base MVOI Opcode Template

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode hi | | | | | | 1 | 0 | 0 | 1 | 1 | 1 | 1 | reg | | |

| opcode lo |
|---|

## Opcode Formats

As bits 3 - 9 of the first word take a fixed value, the table below only considers bits 0 - 2 and 10 - 15 of the first word, and all 16 bits of the second word.

| | First Word Upper 6 | | | | | | First Word Lower 3 | Second Word | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Format** | 15 | | | | | 10 | 2 0 | 15 8 | 7 4 | 3 0 | **Description** |
| MVOI | 0 | 0 | 0 | 0 | 0 | 0 | src1r | imm16 | | | Native MVOI |
| *Reserved* | 0 | 0 | 0 | 0 | 0 | 1 | - | - | | | Reserved |
| r, x, x | 0 | 0 | 0 | 0 | 1 | S | src1r | opcodeA | src2xr | dstxr | 3-register instruction |
| r, p4, x | 0 | 0 | 0 | 1 | 0 | S | src1r | opcodeA | ucst4 | dstxr | Small positive constant |
| r, n4, x | 0 | 0 | 0 | 1 | 1 | S | src1r | opcodeA | ~ucst4 | dstxr | Small negative constant |
| *Reserved* | 0 | 1 | 0 | x | x | x | - | - | | | *Reserved* |
| PSHM PULM | 0 | D | 1 | 0 | 0 | LS | src1r | Register Bitmap | | | Push/pull multiple Not implemented |
| x, x, x | 0 | X1 | 1 | 0 | 1 | 0 | src1xr | opcodeA | src2xr | dstxr | 3-register, all X regs. |
| x, p4, x | 0 | X1 | 1 | 1 | 0 | 0 | src1xr | opcodeA | ucst4 | dstxr | Small positive constant |
| x, n4, x | 0 | X1 | 1 | 1 | 1 | 0 | src1xr | opcodeA | ~ucst4 | dstxr | Small negative constant |
| Long Immediate | 1 | opC | | dstxr | | | src1r | imm16 | | | Long immediate; Not implemented. |

| Term | Meaning |
|---|---|
| src1r | First input operand, one of R0 .. R7 |
| src1xr | First input operand, one of X0 .. XF; combines with X1 field. |
| X1 | Additional bit for src1xr |
| src2xr | Second operand, one of X0 .. XF |
| ucst4 | Unsigned 4-bit constant, 0x0000 .. 0x000F |
| ~ucst4 | Inverted unsigned 4-bit constant, 0xFFFF .. 0xFFF0 |
| dstxr | Destination register, one of X0 .. XF.<br>Some instructions write 32-bit results to dstxr and dstxr + 1. |

| | |
|---|---|
| opcodeA | Opcode from set A. |
| opC | Opcode from set C. |
| S | “Swap”/”Secondary” bit. For non-commutative operations, this swaps src1 and src2. For commutative operations, this selects an alternate, related operation. |
| imm16 | 16-bit immediate value |
| x | Extended register, one of X0 .. XF |
| r | Native register, one of R0 .. R7 |
| p4 | Positive 4-bit constant; equivalent to ucst4. |
| n4 | Negative 4-bit constant; equivalent to ~ucst4. |
| LS | Load/Store. Distinguishes between PULM and PSHM. |
| D | Direction flag. Distinguishes between post-increment and pre-decrement. |

## Operand Types and Encoding

Most Extended Register-to-Register operations accept three operands.

The first operand (src1) is always either a native register (R0 .. R7), or an extended register (X0 .. XF). The first operand is always encoded in bits 0 .. 2 of the first word of the instruction. For extended operands, bit 14 provides an additional bit for the register number.

The second operand (src2) is always either an extended register (X0 .. XF), or a signed 5-bit constant. Due to encoding constraints, the signed 5-bit constant is actually represented as two separate opcode spaces: One with an unsigned 4-bit constant, and one with an inverted 4-bit constant. This is equivalent to a signed 5-bit constant, with the sign bit encoded in bit 11 of the opcode.

The third operand (dst) is always an extended register (X0 .. XF). This register’s value gets copied to PV prior to executing the operation. In some cases, such as the CMPxx& instructions, the destination register also acts as a source operand. For instructions with 32-bit results, CP-1600X writes *two* registers, and dst indicates the lower-numbered register of the pair.

The S bit, which means either Swap or Secondary, modifies the operands as follows:

- For non-commutative operations, such as subtract, setting S = 1 swaps src1 and src2 internally before performing the operation. This enables you to perform both X0 = R1 - X2 and X0 = X2 - R1, for example.
- For commutative operations, such as addition, setting S = 1 selects an alternate, or *secondary* operation. For example, in the case of ADD, S = 1 selects NADD, which performs -(src1 + src2).

The opcode table indicates which opcodes interpret S as Swap vs. Secondary,

## Opcode Set A

At present, this is the only opcode set CP-1600X implements. The following table summarizes the assigned opcodes. See the individual instruction descriptions for detailed information on each.

| Opcode | Mnemonic (S=0) | Mnemonic (S=1) | Description |
|---|---|---|---|
| 00000000 | ADD3 | NADD | Addition. S = 1 negates result. |
| 00000001 | ADDFX | NADDFX | Addition, fixed-point. S = 1 negates result. |
| 00000010 | SUB3 | | Subtraction |
| 00000011 | SUBFX | | Subtraction, fixed-point |
| 00000100 | AND3 | NAND | Bitwise AND. S = 1 inverts result. |
| 00000101 | ANDN | ORN | Bitwise AND w/ src2 negated. S = 1 inverts result. |
| 00000110 | OR3 | NOR | Bitwise OR. S = 1 inverts result. |
| 00000111 | XOR3 | XNOR | Bitwise XOR. S = 1 inverts result. |
| | | | |
| 00001000 | SHL3 | | Shift 16-bit signed value left into 32-bit result |
| 00001001 | SHLU3 | | Shift 16-bit unsigned value left into 32-bit result |
| 00001010 | SHR3 | | Shift 16-bit signed value right into 32-bit result |
| 00001011 | SHRU3 | | Shift 16-bit unsigned value right into 32-bit result |
| 00001100 | BSHLU | | Byte-masked left shift; 2nd reg gets shifted-away bits |
| 00001101 | BSHRU | | Byte-masked right shift; 2nd reg gets shifted-away bits |
| 00001110 | ROL | | Rotate left |
| 00001111 | ROR | | Rotate right |
| | | | |
| 00010000 | BITCNTL / BITCNT[4] | | Count 1 bits in the 'src2 + 1' MSBs of src1 |
| 00010001 | BITCNTR | | Count 1 bits in the 'src2 + 1' LSBs of src1 |
| 00010010 | BITREVL / BITREV[3] | | Bit reverse upper 'src2 + 1' bits of src1 into dst |
| 00010011 | BITREVR | | Bit reverse lower 'src2 + 1' bits of src1 into dst |
| 00010100 | LMO | | Find leftmost 1 in src1 at or below bit # in src2 |
| 00010101 | LMZ | | Find leftmost 0 in src1 at or below bit # in src2 |
| 00010110 | RMO | | Find rightmost 1 in src1 at or above bit # in src2 |
| 00010111 | RMZ | | Find rightmost 0 in src1 at or above bit # in src2 |

[4] BITCNT and BITREV are aliases for BITCNTL and BITREVL that operate on the full word.

| | | | |
|---|---|---|---|
| 00011000 | REPACK | | 32-bit output: PACKH and PACKL |
| 00011001 | PACKL | | dst = (LSB src1) << 8 \| (LSB src2) |
| 00011010 | PACKH | | dst = (MSB src1) << 8 \| (MSB src2) |
| 00011011 | PACKLH | | dst = (LSB src1) << 8 \| (MSB src2) |
| 00011100 | BTOG | | Toggle bit #src2 in src1, writing to dst |
| 00011101 | BSET | | Set bit #src2 in src1, writing to dst |
| 00011110 | BCLR | | Clear bit #src2 in src1, writing to dst |
| 00011111 | CMPEQ | CMPNE | dst = src1 == src2 ? -1 : 0; S=1 inverts result |
| | | | |
| 00100000 | CMPLTU / CMPGTU | | dst = src1 < src2 ? -1 : 0, unsigned |
| 00100001 | CMPLTFXU / CMPGTFXU | | dst = src1 < src2 ? -1 : 0, unsigned fixed-point |
| 00100010 | CMPLEU / CMPGEU | | dst = src1 <= src2 ? -1 : 0, unsigned |
| 00100011 | CMPLEFXU / CMPGEFXU | | dst = src1 <= src2 ? -1 : 0, unsigned fixed-point |
| 00100100 | CMPLTU& / CMPLTU& | | dst &= src1 < src2 ? -1 : 0, unsigned |
| 00100101 | CMPLTFXU& / CMPLTFXU& | | dst &= src1 < src2 ? -1 : 0, unsigned fixed-point |
| 00100110 | CMPLEU& / CMPLEU& | | dst &= src1 <= src2 ? -1 : 0, unsigned |
| 00100111 | CMPLEFXU& / CMPLEFXU& | | dst &= src1 <= src2 ? -1 : 0, unsigned fixed-point |
| | | | |
| 00101000 | CMPLT / CMPGT | | dst = src1 < src2 ? -1 : 0, signed |
| 00101001 | CMPLTFX / CMPGTFX | | dst = src1 < src2 ? -1 : 0, signed fixed-point |
| 00101010 | CMPLE / CMPGE | | dst = src1 <= src2 ? -1 : 0, signed |
| 00101011 | CMPLEFX / CMPGEFX | | dst = src1 <= src2 ? -1 : 0, signed fixed-point |
| 00101100 | CMPLT& / CMPGT& | | dst &= src1 < src2 ? -1 : 0, signed |
| 00101101 | CMPLTFX& / CMPGTFX& | | dst &= src1 < src2 ? -1 : 0, signed fixed-point |
| 00101110 | CMPLE& / CMPGE& | | dst &= src1 <= src2 ? -1 : 0, signed |
| 00101111 | CMPLEFX& / CMPGEFX& | | dst &= src1 <= src2 ? -1 : 0, signed fixed-point |
| | | | |

| | | | |
|---|---|---|---|
| 00110000 | MIN | MINU | dst = min(src1, src2); S determines signedness |
| 00110001 | MINFX | MINFXU | dst = min(src1, src2), fixed-point; S determines signedness |
| 00110010 | MAX | MAXU | dst = max(src1, src2); S determines signedness |
| 00110011 | MAXFX | MAXFXU | dst = max(src1, src2), fixed-point; S determines signedness |
| 00110100 | BOUND | BOUNDU | Bounds dst to the range min(src1,src2) <= dst <= max(src1,src2); S determines signedness |
| 00110101 | BOUNDFX | BOUNDFXU | LIke BOUND, but for fixed-point. |
| 00110110 | ADDCIRC | | Add src1 to dst, updating only src2 LSBs; for circular queues |
| 00110111 | SUBCIRC | | Subtracts src1 from dst, updating only src2 LSBs; for circular queues |
| | | | |
| 00111000 | ATAN2 | | Arctangent of (src2, src1) to one part in 16 |
| 00111001 | ATAN2FX | | Arctangent of (src2, src1) to one part in 16, fixed point |
| 00111010 | SUBABS | SUBABSU | dst = abs(src1 - src2); S determines signedness of inputs |
| 00111011 | SUBABSFX | SUBABSFXU | dst = abs(src1 - src2); S determines signedness ; fixed point |
| 00111100 | DIST | DISTU | Fast distance approximation; S determines signedness |
| 00111101 | DISTFX | DISTFXU | Fast distance approx, fixed-point; S determines signedness. |
| 00111110 | SUMSQ | SUMSQU | Saturated 32-bit unsigned sum of squares; S determines signedness of inputs. Output is not fixed point. |
| 00111111 | SUMSQFX | SUMSQFXU | |
| | | | |
| 01000000 | MPYSS | MPYUU | 32-bit product of 16-bit values |
| 01000001 | MPYFXSS | MPYFXUU | 16-bit product of 16-bit fixed-point values |
| 01000010 | MPYSU | | 32-bit product of 16-bit values |
| 01000011 | MPYFXSU | | 16-bit product of 16-bit fixed-point values |
| 01000100 | MPYUS | | 32-bit product of 16-bit values |
| 01000101 | MPYFXUS | | 16-bit product of 16-bit fixed-point values |
| 01000110 | MPY16 | | 16-bit product of 16-bit values (S/U doesn't matter) |
| 01000111 | ISQRT | ISQRTFX | Square root; S determines integer vs. fixed. |
| | | | |

| | | | |
|---|---|---|---|
| 01001000 | AAL | | ASCII Adjust Lo: (((src1 & 0xFF) - 0x20) << 3) + src2 |
| 01001001 | AAH | | ASCII Adjust Hi: ((((src1 >> 8) & 0xFF) - 0x20) << 3) + src2 |
| 01001010 | DIVS | | 16-bit signed divide; output is rem:quot in 2 regs |
| 01001011 | DIVFXS | | 16-bit signed fixed-point divide; output is rem:quot in 2 regs |
| 01001100 | DIVU | | 16-bit unsigned divide; output is rem:quot in 2 regs |
| 01001101 | DIVFXU | | 16-bit unsigned fixed-point divide; output is rem:quot in 2 regs |
| 01001110 | DIV32S | *reserved* | 32/16 ⟹ 16 signed divide; output is rem:quot in 2 regs |
| 01001111 | DIV32U | *reserved* | 32/16 ⟹ 16 unsigned divide; output is rem:quot in 2 regs |
| | | | |
| 01010000 | ADDS | ADDU | 16 + 16 ⟹ 32 addition with sign or zero extension |
| 01010001 | ADDH | ADDM | ADDH: 16 + 16 + 16 ⟹ 16, with third operand from dst<br>ADDM: unsigned 16 + 16 + 16 ⟹ 32, with third op from dst_lo |
| 01010010 | SUBS | | signed 16 - signed 16 ⟹ 32 |
| 01010011 | SUBU | | unsigned 16 - unsigned 16 ⟹ 32 |
| 01010100 | SUBM | | unsigned 16 - unsigned 16 + signed 16(dst_lo) ⟹ 32 |
| 01010101 | SUBH | | signed 16 - signed 16 + signed 16(dst_lo) ⟹ 32 |
| 01010110 | DMOV | | src1:src2 ⟹ dst_hi:dst_lo |
| 01010111 | ADDSUB | | dst_hi ⟸ src1 + src2; dst_lo ⟸ src1 - src2 |
| | | | |
| 01011000 | ABCD | ABCDL | Adds 2 BCD numbers. 32-bit output if S=1, w/ carry in dst_hi |
| 01011001 | ABCDH | ABCDM | ABCDH: Adds 2 BCD numbers + 3-bit carry/borrow in dst.<br>ABCDM: Like ABCDH, with 3-bit carry/borrow written to dst_hi. |
| 01011010 | SBCD | | Subtracts 2 BCD numbers; 16-bit result |
| 01011011 | SBCDL | | Subtracts 2 BCD numbers; dst_hi gets 3-bit carry/borrow |
| 01011100 | SBCDM | | Subtracts 2 BCD numbers + 3-bit carry/borrow in dst_lo; dst_hi gets 3-bit carry/borrow result. |
| 01011101 | SBCDH | | Subtracts 2 BCD numbers + 3-bit carry/borrow in dst; 16-bit rslt |
| 01011110 | I2BCD | | Convert 32-bit int to 32-bit BCD; clamps to 99999999 |
| 01011111 | BCD2I | | Convert 32-bit BCD to 32-bit int |
| | | | |
| 01100000 | CMPEQ& | CMPNE& | dst &= src1 == src2 ? -1 : 0; S = 1 inverts output |
| 01100001 .. 11111111 | | | Reserved |

## Opcode Set B

This opcode set currently does not exist.

## Opcode Set C

This opcode set exists for the *currently unimplemented* long-immediate opcode format.

This opcode space has room for four instructions. While this opcode space remains reserved, this specification proposes the following four opcode assignments:

| Opcode | Meaning |
|---|---|
| 00 | ADD. dstxr = src1r + immediate |
| 01 | AND. dstxr = src1r & immediate |
| 10 | OR. dstxr = src1r \| immediate |
| 11 | MERGE. dstxr = (src1r & immediate) \| (dstxr & ~immediate) |

## Instruction Descriptions

### ADD3, opcodeA = 00000000b, S = 0

Adds two 16-bit numbers together, producing a 16-bit result.

```
dst = src1 + src2
```

### NADD, opcodeA = 00000000b, S = 1

Adds two 16-bit numbers together and negates the sum, producing a 16-bit result.

```
dst = -(src1 + src2)
```

### ADDFX, opcodeA = 00000001b, S = 0

Adds two 16-bit fixed-point numbers together, producing a 16-bit fixed-point result.

```
dst = swap[5](swap(src1) + swap(src2))
```

### NADDFX, opcodeA = 00000001b, S = 1

Adds two 16-bit fixed-point numbers together and negates the sum, producing a 16-bit fixed-point result.

```
dst = swap(-(swap(src1) + swap(src2))
```

### SUB3, opcodeA = 00000010b

Subtracts two 16-bit numbers, producing a 16-bit result.

```
dst = src1 - src2
```

### SUBFX, opcodeA = 00000011b

Subtracts two 16-bit numbers, producing a 16-bit result.

```
dst = swap(swap(src1) - swap(src2))
```

### AND3, opcodeA = 00000100b, S = 0

Computes the bitwise-AND between src1 and src2, producing a 16-bit result.

```
dst = src1 & src2
```

### NAND, opcodeA = 00000100b, S = 1

Computes the inverse of the bitwise-AND between src1 and src2, producing a 16-bit result.

```
dst = ~(src1 & src2)
```

[5] The `swap()` primitive swaps the upper and lower bytes of a 16-bit number, similar to the CP-1600 SWAP instruction.

### ANDN, opcodeA = 00000101b, S = 0

Computes the bitwise-AND between src1 and the inverse of src2, producing a 16-bit result.

```
dst = src1 & ~src2;
```

### ORN, opcodeA = 00000101b, S = 1

Computes the inverse of the bitwise-AND between the src2 and inverse of src1, producing a 16-bit result. This is equivalent to computing the bitwise-OR of src 2 with the inverse of src1.

```
dst = ~(src1 & ~src2);
dst = (~src1) | src2;     //  Alternate interpretation, applying deMorgan's Law
```

### OR3, opcodeA = 00000110b, S = 0

Computes the bitwise-OR between src1 and src2, producing a 16-bit result.

```
dst = src1 | src2;
```

### NOR, opcodeA = 00000110b, S = 1

Computes the inverse of the bitwise-OR between src1 and src2, producing a 16-bit result.

```
dst = ~(src1 | src2);
```

### XOR3, opcodeA = 00000111b, S = 0

Computes the bitwise-XOR between src1 and src2, producing a 16-bit result.

```
dst = src1 ^ src2;
```

### XNOR, opcodeA = 00000111b, S = 1

Computes the inverse of the bitwise-XOR between src1 and src2, producing a 16-bit result.

```
dst = ~(src1 ^ src2);
```

### SHL3, opcodeA = 00001000b

Shifts src1 left by the number of bit positions specified in src2[3:0]. Value is first sign extended to 32 bits before shifting. The 32-bit result is placed in a pair of registers. You can also think of dst_lo as capturing the result of a 16-bit shift, and dst_hi as capturing the "shifted away" bits.

```
dst_hi:dst_lo = sign_extend32(src1) << (src2 & 0xF);
```

### SHLU3, opcodeA = 00001001b

Shifts src1 left by the number of bit positions specified in src2[3:0]. Value is first zero extended to 32 bits before shifting. The 32-bit result is placed in a pair of registers. You can also think of dst_lo as capturing the result of a 16-bit shift, and dst_hi as capturing the "shifted away" bits..

```
dst_hi:dst_lo = zero_extend32(src1) << (src2 & 0xF);
```

### SHR3, opcodeA = 00001010b

Shifts src1 right by the number of bit positions specified in src2[3:0]. Value is first deposited in the upper 16 bits of a 32-bit signed value before shifting. The right shift performs sign-extension. The 32-bit result is placed in a pair of registers, dst_lo:dst_hi. You can also think of dst_lo as capturing the result of a 16-bit shift, and dst_hi as capturing the "shifted away" bits.

```
dst_lo:dst_hi = signed32(src1 << 16) >> (src2 & 0xF);
```

### SHRU3, opcodeA = 00001011b

Shifts src1 right by the number of bit positions specified in src2[3:0]. Value is first deposited in the upper 16 bits of a 32-bit unsigned value before shifting. The right shift performs zero-extension. The 32-bit result is placed in a pair of registers, dst_lo:dst_hi. You can also think of dst_lo as capturing the result of a 16-bit shift, and dst_hi as capturing the "shifted away" bits.

```
dst_lo:dst_hi = unsigned32(src1 << 16) >> (src2 & 0xF);
```

### BSHLU, opcodeA = 00001100b

Byte-masked shifts shift each byte within a 16-bit word independently. BSHLU shifts each byte in src1 left by the amount specified in src2[3:0], clamped to the range 0..8. It writes the shifted result to dst_lo, and the "shifted away" bits to dst_hi. This is intended for manipulating byte-oriented graphics packed into 16-bit words.

```
src1_lsb = src1 & 0xFF;
src1_msb = (src1 >> 8) & 0xFF;
shift = (src2 & 0xF) < 8 ? src2 & 0xF : 8;
shifted_lsb = (src1_lsb << shift);
shifted_msb = (src2_msb << shift);
dst_lo = ((shifted_msb & 0xFF) << 8) | (shifted_lsb & 0xFF);
dst_hi = (shifted_msb & 0xFF00) | ((shifted_lsb >> 8) & 0xFF);
```

### BSHRU, opcodeA = 00001101b

Byte-masked shifts shift each byte within a 16-bit word independently. BSHRU shifts each byte in src1 right by the amount specified in src2[3:0], clamped to the range 0..8. It writes the shifted result to dst_lo, and the "shifted away" bits to dst_hi. This is intended for manipulating byte-oriented graphics packed into 16-bit words.

```
src1_lsb = (src1 & 0xFF) << 8;
src1_msb = src1 & 0xFF00;
shift = (src2 & 0xF) < 8 ? src2 & 0xF : 8;
shifted_lsb = (src1_lsb >> shift);
shifted_msb = (src2_msb >> shift);
dst_lo = (shifted_msb & 0xFF00) | ((shifted_lsb >> 8) & 0xFF);
dst_hi = ((shifted_msb & 0xFF) << 8) | (shifted_lsb & 0xFF);
```

### ROL, opcodeA = 00001110b

Rotates the bits in src1 left by the number of bit positions specified in src2[3:0].

```
shift = src2 & 0xF;
dst = (src1 << shift) | (src1 >> (16 - shift));
```

### ROR, opcodeA = 00001111b

Rotates the bits in src1 right by the number of bit positions specified in src2[3:0].

```
shift = src2 & 0xF;
dst = (src1 >> shift) | (src1 << (16 - shift));
```

### BITCNTL, opcodeA = 00010000b

Counts the number of 1 bits in src1, looking only at src2[3:0] + 1 bits starting from the left. For example, `BITCNTL R0, 3, X0` will examine bits 15, 14, and 13.

```
to_count = (src2 & 0xF) + 1;
one_bits = 0;
mask = 0x8000u;
while (to_count != 0) {
  if (src1 & mask) {
    one_bits++;
  }
  to_count--;
  mask >>= 1;
}
dst = one_bits;
```

### BITCNTR, opcodeA = 00010000b

Counts the number of 1 bits in src1, looking only at src2[3:0] + 1 bits starting from the right. For example, `BITCNTR R0, 3, X0` will examine bits 0, 1, and 2.

```
to_count = (src2 & 0xF) + 1;
one_bits = 0;
mask = 0x0001u;
while (to_count != 0) {
  if (src1 & mask) {
    one_bits++;
  }
  to_count--;
  mask <<= 1;
}
dst = one_bits;
```

### BITREVL, opcodeA = 00010010b

Performs a bit-reversal on the left-most bits of src1, writing the result to dst. Only reverses src2[3:0]+1 bits at the left. Other bits in the result are zeroed. Can be useful for flipping a bitmapped graphic.

Example: `BITREVL R0, 3, X0` will write bits 15, 14, and 13 of `R0` to bits 13, 14, and 15 of `X0`, while the rest of `X0` will be filled with zeros.

```
to_reverse = (src2 & 0xF) + 1;
value_in = src1;
value_out = 0;

while (to_reverse != 0) {
  value_out >>= 1;
  if (value_in & 0x8000) {
    value_out |= 0x8000;
  }
  value_in <<= 1;
  to_reverse--;
}

dst = value_out;
```

### BITREVR, opcodeA = 00010011b

Performs a bit-reversal on the right-most bits of src1, writing the result to dst. Only reverses src2[3:0]+1 bits at the right. Other bits in the result are zeroed. Can be useful for flipping a bitmapped graphic.
Example: `BITREVR R0, 3, X0` will write bits 0, 1, and 2 of `R0` to bits 2, 1, and 0 of `X0`, while the rest of `X0` will be filled with zeros.

```
to_reverse = (src2 & 0xF) + 1;
value_in = src1;
value_out = 0;

while (to_reverse != 0) {
  value_out <<= 1;
  if (value_in & 0x0001) {
    value_out |= 0x0001;
  }
  value_in >>= 1;
  to_reverse--;
}

dst = value_out;
```

### LMO, opcodeA = 00010100b

Left-Most One: Locates the left-most 1 bit in src1, at or below the bit number specified in src2[3:0]. Returns the bit position of the 1 bit, or -1 if no 1 bit is found.

```
one_bit = -1;
for (i = src2 & 0xF; i >= 0; i--) {
  if ((src1 & (1 << i)) != 0) {
    one_bit = i;
    break;
  }
}
dst = one_bit;
```

### LMZ, opcodeA = 00010101b

Left-Most Zero: Locates the left-most 0 bit in src1, at or below the bit number specified in src2[3:0]. Returns the bit position of the 0 bit, or -1 if no 0 bit  is found.

```
zero_bit = -1;
for (i = src2 & 0xF; i >= 0; i--) {
  if ((src1 & (1 << i)) == 0) {
    zero_bit = i;
    break;
  }
}
dst = zero_bit;
```

### RMO, opcodeA = 00010110b

Right-Most One: Locates the right-most 1 bit in src1, at or above the bit number specified in src2[3:0]. Returns the bit position of the 1 bit, or -1 if no 1 bit is found.

```
one_bit = -1;
for (i = src2 & 0xF; i <= 15; i++) {
  if ((src1 & (1 << i)) != 0) {
    one_bit = i;
    break;
  }
}
dst = one_bit;
```

### RMZ, opcodeA = 00010111b

Right-Most Zero: Locates the left-most 0 bit in src1, at or above the bit number specified in src2[3:0]. Returns the bit position of the 0 bit, or -1 if no 0 bit is found.

```
zero_bit = -1;
for (i = src2 & 0xF; i <= 15; i++) {
  if ((src1 & (1 << i)) == 0) {
    zero_bit = i;
    break;
  }
}
dst = zero_bit;
```

### REPACK, opcodeA = 00011000b

Packs the lower bytes of src1 and src2 into one destination register, and the upper bytes of src1 and src2 into another.

```
dst_lo = ((src1 & 0xFF) << 8) | (src2 & 0xFF);
dst_hi = (src1 & 0xFF00) | ((src2 >> 8) & 0xFF);
```

### PACKL, opcodeA = 00011001b

Packs the lower bytes of src1 and src2 into one destination register. It's the dst_lo half of REPACK.

```
dst = ((src1 & 0xFF) << 8) | (src2 & 0xFF);
```

### PACKH, opcodeA = 00011010b

Packs the upper bytes of src1 and src2 into one destination register. It's the dst_hi half of REPACK.

```
dst = (src1 & 0xFF00) | ((src2 >> 8) & 0xFF);
```

### PACKLH, opcodeA = 00011011b

Packs the lower byte of src1 and upper byte of src2 into one destination register. Useful for extracting the middle 16 bits out of a 32-bit quantity.

```
dst = ((src1 & 0xFF) << 8) | ((src2 >> 8) & 0xFF);
```

### BTOG, opcodeA = 00011100b

Toggles the bit in src1 specified by src2[3:0], and writes the result to dst.

```
dst = src1 ^ (1u << (src2 & 0xF));
```

### BSET, opcodeA = 00011101b

Sets the bit in src1 specified by src2[3:0], and writes the result to dst.

```
dst = src1 | (1u << (src2 & 0xF));
```

### BCLR, opcodeA = 00011110b

Clears the bit in src1 specified by src2[3:0], and writes the result to dst.

```
dst = src1 & ~(1u << (src2 & 0xF));
```

### CMPEQ, opcodeA = 00011111b, S = 0

Sets dst to 0 or -1 (0xFFFF) based on whether src1 equals src2.

```
dst = src1 == src2 ? -1 : 0;
```

### CMPNE, opcodeA = 00011111b, S = 1

Sets dst to 0 or -1 (0xFFFF) based on whether src1 equals src2.

```
dst = src1 != src2 ? -1 : 0;
```

### CMPLTU / CMPGTU, opcodeA = 00100000b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as unsigned integers. `CMPGTU` is an alias for `CMPLTU` with its operands swapped.

```
dst = unsigned(src1) < unsigned(src2) ? -1 : 0;
```

### CMPLTFXU / CMPGTFXU, opcodeA = 00100001b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as unsigned fixed-point numbers. CMPGTFXU is an alias for CMPLTFXU with its operands swapped.

```
dst = unsigned(swap(src1)) < unsigned(swap(src2)) ? -1 : 0;
```

### CMPLEU / CMPGEU, opcodeA = 00100010b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as unsigned integers. CMPGEU is an alias for CMPLEU with its operands swapped.

```
dst = unsigned(src1) <= unsigned(src2) ? -1 : 0;
```

### CMPLEFXU / CMPGEFXU, opcodeA = 00100011b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as unsigned fixed-point numbers. CMPGEFXU is an alias for CMPLEFXU with its operands swapped.

```
dst = unsigned(swap(src1)) <= unsigned(swap(src2)) ? -1 : 0;
```

### CMPLTU& / CMPGTU&, opcodeA = 00100100b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as unsigned integers. Useful for creating compound conditions or masking values based on a condition. CMPGTU& is an alias for CMPLTU& with its operands swapped.

```
dst &= unsigned(src1) < unsigned(src2) ? -1 : 0;
```

### CMPLTFXU& / CMPGTFXU&, opcodeA = 00100101b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as unsigned fixed-point numbers. Useful for creating compound conditions or masking values based on a condition. CMPGTFXU& is an alias for CMPLTFXU& with its operands swapped.

```
dst &= unsigned(swap(src1)) < unsigned(swap(src2)) ? -1 : 0;
```

### CMPLEU& / CMPGEU&, opcodeA = 00100110b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as unsigned integers. Useful for creating compound conditions or masking values based on a condition. CMPGEU& is an alias for CMPLEU& with its operands swapped.

```
dst &= unsigned(src1) <= unsigned(src2) ? -1 : 0;
```

### CMPLEFXU& / CMPGEFXU&, opcodeA = 00100111b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as unsigned fixed-point numbers. Useful for creating compound conditions or masking values based on a condition. CMPGEFXU& is an alias for CMPLEFXU& with its operands swapped.

```
dst &= unsigned(swap(src1)) <= unsigned(swap(src2)) ? -1 : 0;
```

### CMPLT / CMPGT, opcodeA = 00101000b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as signed integers. CMPGT is an alias for CMPLT with its operands swapped.

```
dst = signed(src1) < signed(src2) ? -1 : 0;
```

### CMPLTFX / CMPGTFX, opcodeA = 00101001b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as signed fixed-point numbers. CMPGTFX is an alias for CMPLTFX with its operands swapped.

```
dst = signed(swap(src1)) < signed(swap(src2)) ? -1 : 0;
```

### CMPLE / CMPGE, opcodeA = 00101010b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as signed integers. CMPGE is an alias for CMPLE with its operands swapped.

```
dst = signed(src1) <= signed(src2) ? -1 : 0;
```

### CMPLEFX / CMPGEFX, opcodeA = 00101011b

Sets dst to 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as signed fixed-point numbers. CMPGEFX is an alias for CMPLEFX with its operands swapped.

```
dst = signed(swap(src1)) <= signed(swap(src2)) ? -1 : 0;
```

### CMPLT& / CMPGT&, opcodeA = 00101100b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as signed integers. Useful for creating compound conditions or masking values based on a condition. CMPGT& is an alias for CMPLT& with its operands swapped.

```
dst &= signed(src1) < signed(src2) ? -1 : 0;
```

### CMPLTFX& / CMPGTFX&, opcodeA = 00101101b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than src2, interpreting them as signed fixed-point numbers. Useful for creating compound conditions or masking values based on a condition. CMPGTFX& is an alias for CMPLTFX& with its operands swapped.

```
dst &= signed(swap(src1)) < signed(swap(src2)) ? -1 : 0;
```

### CMPLE& / CMPGE&, opcodeA = 00101110b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as signed integers. Useful for creating compound conditions or masking values based on a condition. CMPGE& is an alias for CMPLE& with its operands swapped.

```
dst &= signed(src1) <= signed(src2) ? -1 : 0;
```

### CMPLEFX& / CMPGEFX&, opcodeA = 00101111b

Bitwise ANDs dst with 0 or -1 (0xFFFF) based on whether src1 is less than or equal to src2, interpreting them as signed fixed-point numbers. Useful for creating compound conditions or masking values based on a condition. CMPGEFX& is an alias for CMPLEFX& with its operands swapped.

```
dst &= signed(swap(src1)) <= signed(swap(src2)) ? -1 : 0;
```

MIN, opcodeA = 00110000b, S = 0

Writes the minimum of src1 and src2 to dst, treating the inputs as signed integers.

```
dst = signed(src1) < signed(src2) ? src1 : src2;
```

MINU, opcodeA = 00110000b, S = 1

Writes the minimum of src1 and src2 to dst, treating the inputs as unsigned integers.

```
dst = unsigned(src1) < unsigned(src2) ? src1 : src2;
```

MINFX, opcodeA = 00110001b, S = 0

Writes the minimum of src1 and src2 to dst, treating the inputs as signed fixed-point values.

```
dst = signed(swap(src1)) < signed(swap(src2)) ? src1 : src2;
```

MINFXU, opcodeA = 00110001b, S = 1

Writes the minimum of src1 and src2 to dst, treating the inputs as unsigned fixed-point values.

```
dst = unsigned(swap(src1)) < unsigned(swap(src2)) ? src1 : src2;
```

MAX, opcodeA = 00110010b, S = 0

Writes the maximum of src1 and src2 to dst, treating the inputs as signed integers.

```
dst = signed(src1) > signed(src2) ? src1 : src2;
```

MAXU, opcodeA = 00110010b, S = 1

Writes the maximum of src1 and src2 to dst, treating the inputs as unsigned integers.

```
dst = unsigned(src1) > unsigned(src2) ? src1 : src2;
```

### MAXFX, opcodeA = 00110011b, S = 0

Writes the maximum of src1 and src2 to dst, treating the inputs as signed fixed-point values.

```
dst = signed(swap(src1)) > signed(swap(src2)) ? src1 : src2;
```

### MAXFXU, opcodeA = 00110011b, S = 1

Writes the maximum of src1 and src2 to dst, treating the inputs as unsigned fixed-point values.

```
dst = unsigned(swap(src1)) > unsigned(swap(src2)) ? src1 : src2;
```

### BOUND, opcodeA = 00110100b, S = 0

Clamps the value in dst to the range min(src1, src2) ≤ dst ≤ max(src1, src2), interpreting the values as signed integers.

```
bound_lo = signed(src1) < signed(src2) ? src1 : src2;
bound_hi = signed(src1) > signed(src2) ? src1 : src2;
val = signed(dst);
dst = val < bound_lo ? bound_lo : val > bound_hi ? bound_hi : val;
```

### BOUNDU, opcodeA = 00110100b, S = 1

Clamps the value in dst to the range min(src1, src2) ≤ dst ≤ max(src1, src2), interpreting the values as unsigned integers.

```
bound_lo = unsigned(src1) < unsigned(src2) ? src1 : src2;
bound_hi = unsigned(src1) > unsigned(src2) ? src1 : src2;
val = unsigned(dst);
dst = val < bound_lo ? bound_lo : val > bound_hi ? bound_hi : val;
```

### BOUNDFX, opcodeA = 00110101b, S = 0

Clamps the value in dst to the range min(src1, src2) ≤ dst ≤ max(src1, src2), interpreting the values as signed fixed-point numbers.

```
in1 = signed(swap(src1));
in2 = signed(swap(src2));
bound_lo = in1 < in2 ? in1 : in2;
bound_hi = in1 > in2 ? in1 : in2;
val = signed(swap(dst));
out = val < bound_lo ? bound_lo : val > bound_hi ? bound_hi : val;
dst = swap(out);
```

### BOUNDFXU, opcodeA = 00110101b, S = 1

Clamps the value in dst to the range min(src1, src2) ≤ dst ≤ max(src1, src2), interpreting the values as signed fixed-point numbers.

```
in1 = unsigned(swap(src1));
in2 = unsigned(swap(src2));
bound_lo = in1 < in2 ? in1 : in2;
bound_hi = in1 > in2 ? in1 : in2;
val = unsigned(swap(dst));
out = val < bound_lo ? bound_lo : val > bound_hi ? bound_hi : val;
dst = swap(out);
```

### ADDCIRC, opcodeA = 00110110b

Adds src1 to dst, updating only the number of LSBs specified in src2[3:0]. This is intended for indexing circular queues.

```
update_mask = (1u << (src2 & 0xF)) - 1;
keep_mask = ~update_mask;
dst = (dst & keep_mask) | ((dst + src1) & update_mask);
```

### SUBCIRC, opcodeA = 00110111b

Subtracts src1 from dst, updating only the number of LSBs specified in src2[3:0]. This is intended for indexing circular queues.

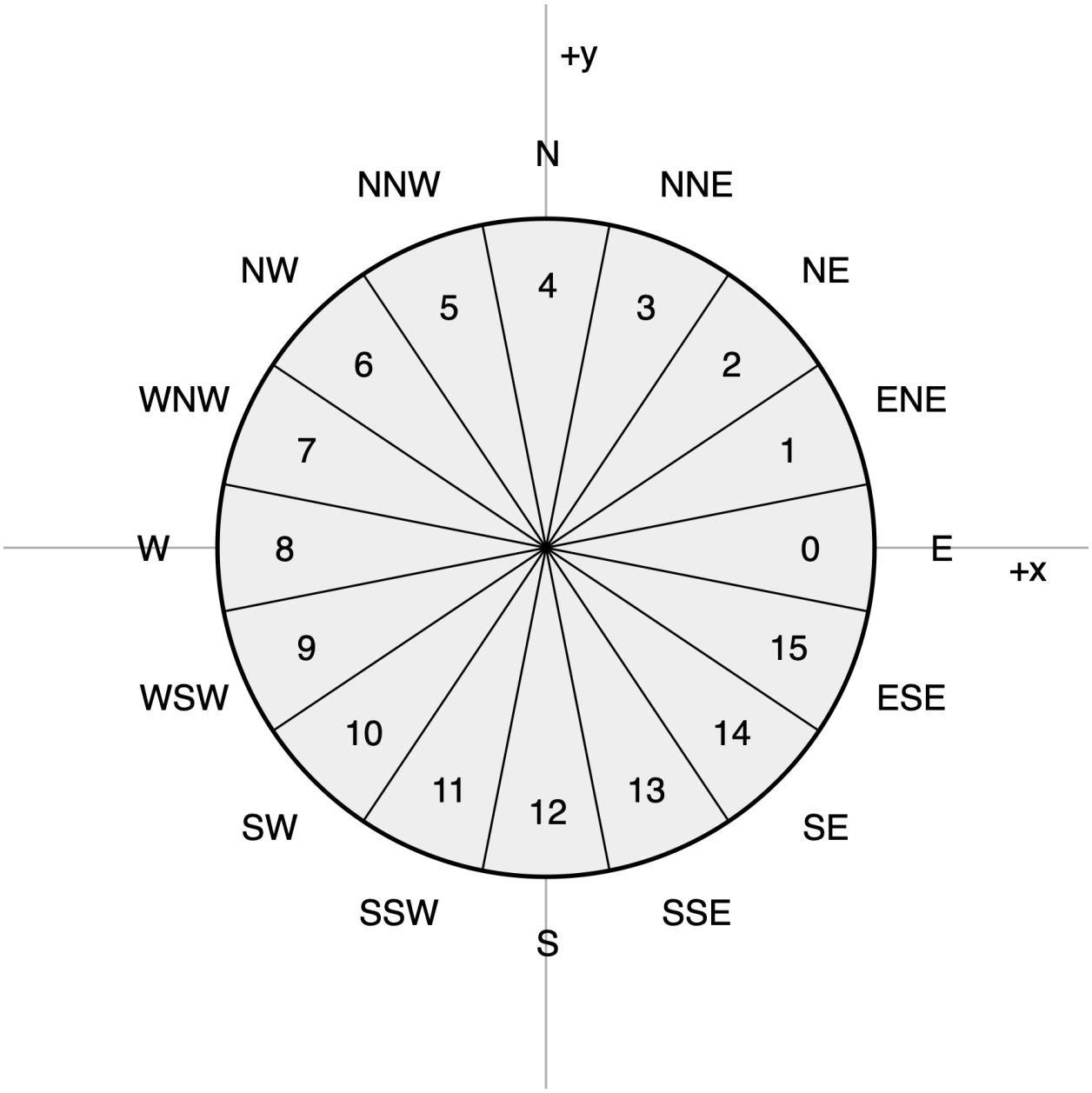
```
update_mask = (1u << (src2 & 0xF)) - 1;
keep_mask = ~update_mask;
dst = (dst & keep_mask) | ((dst - src1) & update_mask);
```

ATAN2, opcodeA = 00111000b

Computes the heading associated with the vector (src1, src2), where src1 is the signed integer X component and src2 is the signed integer Y component. Returns a value 0 through 15, with 0 corresponding to the positive X direction. This is similar to the `atan2()` library function in many environments, modified to map angles onto [0, 15] rather than [-π, π]. The input (0,0) returns 0.

The 16 output values correspond to compass headings as follows, and are intended to be consistent with Intellivision controller input decoding.

| **Output** | **Heading** | **Output** | **Heading** | **Output** | **Heading** | **Output** | **Heading** |
|---|---|---|---|---|---|---|---|
| 0 | E | 4 | N | 8 | W | 12 | S |
| 1 | ENE | 5 | NNW | 9 | WSW | 13 | SSE |
| 2 | NE | 6 | NW | 10 | SW | 14 | SE |
| 3 | NNE | 7 | WNW | 11 | SSW | 15 | ESE |




```
radians = atan2(signed(src2), signed(src1));
if (radians < 0) radians = 2*π - radians;   // Make angle go [0, 2π]
half_sector = int(radians * 16 / π);        // Half sector goes 0 to 31.
sector = ((half_sector + 1) / 2) % 16;      // Map to 0..15, centered on each heading
```

ATAN2FX, opcodeA = 00111001b

Identical to ATAN2, except that it interprets its inputs as signed fixed-point numbers. See ATAN2, opcodeA = 00111000b, above.

### SUBABS, opcodeA = 00111010b, S = 0

Computes the absolute value of (src1 - src2), interpreting both inputs as signed integers.

```
op1 = signed16(src1);
op2 = signed16(src2);
dst = unsigned16(op1 > op2 ? op1 - op2 : op2 - op1);
```

### SUBABSU, opcodeA = 00111010b, S = 1

Computes the absolute value of (src1 - src2), interpreting both as unsigned integers.

```
op1 = unsigned16(src1);
op2 = unsigned16(src2);
dst = unsigned16(op1 > op2 ? op1 - op2 : op2 - op1);
```

### SUBABSFX, opcodeA = 00111011b, S = 0

Computes the absolute value of (src1 - src2), interpreting both as signed fixed-point numbers. Result is unsigned fixed-point.

```
op1 = signed16(swap(src1));
op2 = signed16(swap(src2));
dst = swap(unsigned16(op1 > op2 ? op1 - op2 : op2 - op1));
```

### SUBABSFXU, opcodeA = 00111011b, S = 1

Computes the absolute value of (src1 - src2), interpreting both as unsigned fixed-point numbers. Result is unsigned fixed-point.

```
op1 = unsigned16(swap(src1));
op2 = unsigned16(swap(src2));
dst = swap(unsigned16(op1 > op2 ? op1 - op2 : op2 - op1));
```

### DIST, opcodeA = 00111100b, S = 0

Computes the fast distance approximation from Graphics Gems IV. This has a maximum error of about 4%. Interprets inputs as signed integers and computes distance based on their absolute values.

```
op1 = abs(signed16(src1));
op2 = abs(signed16(src2));
dst = (123 * max(op1, op2) + 51 * min(op1, op2)) / 128;
```

### DISTU, opcodeA = 00111100b, S = 1

Computes the fast distance approximation from Graphics Gems IV. This has a maximum error of about 4%. Interprets inputs as unsigned integers.

```
op1 = unsigned16(src1);
op2 = unsigned16(src2);
dst = (123 * max(op1, op2) + 51 * min(op1, op2)) / 128;
```

### DISTFX, opcodeA = 00111101b, S = 0

Computes the fast distance approximation from Graphics Gems IV. This has a maximum error of about 4%. Interprets inputs as signed fixed-point values and computes distance based on their absolute values.

```
op1 = abs(signed16(swap(src1)));
op2 = abs(signed16(swap(src2)));
dst = swap((123 * max(op1, op2) + 51 * min(op1, op2)) / 128);
```

### DISTFXU, opcodeA = 00111101b, S = 1

Computes the fast distance approximation from Graphics Gems IV. This has a maximum error of about 4%. Interprets inputs as unsigned fixed-point values.

```
op1 = unsigned16(swap(src1));
op2 = unsigned16(swap(src2));
dst = swap((123 * max(op1, op2) + 51 * min(op1, op2)) / 128);
```

SUMSQ, opcodeA = 00111110b, S = 0

Computes the sum of the squares of src1 and src2, clamping to a 32-bit unsigned sum. Both src1 and src2 are interpreted as 16-bit signed integer values.

```
op1 = signed32(src1);
op2 = signed32(src2);
sq = unsigned64(src1*src1 + src2*src2);
dst_hi:dst_lo = min(0xFFFFFFFFu, sq);
```

SUMSQU, opcodeA = 00111110b, S = 1

Computes the sum of the squares of src1 and src2, clamping to a 32-bit unsigned sum. Both src1 and src2 are interpreted as 16-bit unsigned integer values.

```
op1 = unsigned32(src1);
op2 = unsigned32(src2);
sq = unsigned64(src1*src1 + src2*src2);
dst_hi:dst_lo = min(0xFFFFFFFFu, sq);
```

SUMSQFX, opcodeA = 00111111b, S = 0

Computes the sum of the squares of src1 and src2, clamping to a 32-bit unsigned sum. Both src1 and src2 are interpreted as 16-bit signed fixed-point values. Sum is not byte/word swapped. dst_hi holds the integer portion and dst_lo holds the fractional portion.

```
op1 = signed32(swap(src1));
op2 = signed32(swap(src2));
sq = unsigned64(src1*src1 + src2*src2);
dst_hi:dst_lo = min(0xFFFFFFFFu, sq);
```

SUMSQFXU, opcodeA = 00111111b, S = 1

Computes the sum of the squares of src1 and src2, clamping to a 32-bit unsigned sum. Both src1 and src2 are interpreted as 16-bit unsigned fixed-point values. Sum is not byte/word swapped. dst_hi holds the integer portion and dst_lo holds the fractional portion.

```
op1 = signed32(swap(src1));
op2 = signed32(swap(src2));
sq = unsigned64(src1*src1 + src2*src2);
```

```
dst_hi:dst_lo = min(0xFFFFFFFFu, sq);
```

### MPYSS, opcodeA = 01000000b, S = 0

Computes the 32-bit product of src1 and src2, interpreting both as signed integers.

```
dst_hi:dst_lo = signed32(src1) * signed32(src2);
```

### MPYUU, opcodeA = 01000000b, S = 1

Computes the 32-bit product of src1 and src2, interpreting both as unsigned integers.

```
dst_hi:dst_lo = unsigned32(src1) * unsigned32(src2);
```

### MPYFXSS, opcodeA = 01000001b, S = 0

Computes the 16-bit fixed-point product of src1 and src2, interpreting both as signed fixed-point values.

```
op1 = signed32(swap(src1));
op2 = signed32(swap(src2));
prd = ((op1 * op2) >> 8) & 0xFFFF;
dst = swap(prd);
```

### MPYFXUU, opcodeA = 01000001b, S = 1

Computes the 16-bit fixed-point product of src1 and src2, interpreting both as unsigned fixed-point values.

```
op1 = unsigned32(swap(src1));
op2 = unsigned32(swap(src2));
prd = ((op1 * op2) >> 8) & 0xFFFF;
dst = swap(prd);
```

### MPYSU, opcodeA = 01000010b

Computes the 32-bit product of src1 and src2, interpreting src1 as a signed integer, and src2 as an unsigned integer.

```
dst_hi:dst_lo = signed32(src1) * unsigned32(src2);
```

### MPYFXSU, opcodeA = 01000011b

Computes the 16-bit fixed-point product of src1 and src2, interpreting src1 as a signed fixed-point value, and src2 as an unsigned fixed-point value.

```
op1 = signed32(swap(src1));
op2 = unsigned32(swap(src2));
prd = ((op1 * op2) >> 8) & 0xFFFF;
dst = swap(prd);
```

### MPYUS, opcodeA = 01000100b

Computes the 32-bit product of src1 and src2, interpreting src1 as an unsigned integer, and src2 as a signed integer.

```
dst_hi:dst_lo = unsigned32(src1) * signed32(src2);
```

### MPYFXUS, opcodeA = 01000101b

Computes the 16-bit fixed-point product of src1 and src2, interpreting src1 as an unsigned fixed-point value, and src2 as a signed fixed-point value.

```
op1 = unsigned32(swap(src1));
op2 = signed32(swap(src2));
prd = ((op1 * op2) >> 8) & 0xFFFF;
dst = swap(prd);
```

### MPY16, opcodeA = 01000110b

Computes the 16-bit product of src1 and src2, interpreting src1 as integers of unspecified sign.[6]

```
dst = src1 * src2;
```

[6] Should have defined this only for S = 0, so that S = 1 could have some other meaning. Ah well...

### ISQRT, opcodeA = 01000111b, S = 0

Computes the integer square root of src1, interpreting src1 as an unsigned integer. Ignores src2

```
dst = floor(sqrt(unsigned(src1)));
```

### ISQRTFX, opcodeA = 01000111b, S = 1

Computes the fixed-point square root of src1, interpreting src1 as an unsigned fixed-point value. Ignores src2

```
dst = swap(floor(sqrt(unsigned(swap(src1)) * 256)));
```

### AAL, opcodeA = 01001000b

ASCII Adjust Lo: Adjusts the low byte of src1 for display in BACKTAB, assuming it holds an ASCII character. It subtracts 0x20 from the value, shifts left by 3 positions, and then adds a display format adjustment from src2.

```
dst = (((src1 & 0xFF) - 0x20) << 3) + src2;
```

### AAH, opcodeA = 01001001b

ASCII Adjust Hi: Adjusts the high byte of src1 for display in BACKTAB, assuming it holds an ASCII character. It subtracts `0x20` from the value, shifts left by 3 positions, and then adds a display format adjustment from src2.

```
dst = ((((src1 >> 8) & 0xFF) - 0x20) << 3) + src2;
```

### DIVS, opcodeA = 01001010b

Divides src1 by src2, treating both as signed integers. Places the signed quotient in dst_lo and remainder in dst_hi. If src2 is 0, both quotient and remainder get 0x7FFF. Division rounds towards 0, so (-1)/2 = 0 and (-1)%2 = -1.

```
if (src2) {
  dst_lo = signed(src1) / signed(src2);
  dst_hi = signed(src1) % signed(src2);
```

```
} else {
  dst_lo = 0x7FFF;
  dst_hi = 0x7FFF;
}
```

### DIVFXS, opcodeA = 01001011b

Divides src1 by src2, treating both as signed fixed-point values. Places the signed fixed-point quotient in dst_lo and remainder in dst_hi. If src2 is 0, or the divide result is out of range, both quotient and remainder get 0xFF7F.

```
op1 = signed32(swap(src1)) * 256;
op2 = signed32(swap(src2));
dst_lo = swap(0x7FFF);
dst_hi = swap(0x7FFF);
if (op2) {
  quo = op1 / op2;
  rem = op1 % op2;
  if (quo >= -0x8000 && quo <= 0x7FFF) {
    dst_lo = swap(quo);
    dst_hi = swap(rem);
  }
}
```

### DIVU, opcodeA = 01001100b

Divides src1 by src2, treating both as unsigned integers. Places the unsigned quotient in dst_lo and remainder in dst_hi. If src2 is 0, both quotient and remainder get 0xFFFF.

```
if (src2) {
  dst_lo = unsigned(src1) / unsigned(src2);
  dst_hi = unsigned(src1) % unsigned(src2);
} else {
  dst_lo = 0xFFFF;
  dst_hi = 0xFFFF;
}
```

### DIVFXU, opcodeA = 01001101b

Divides src1 by src2, treating both as unsigned fixed-point values. Places the unsigned fixed-point quotient in dst_lo and remainder in dst_hi. If src2 is 0, or the divide result is out of range, both quotient and remainder get 0xFFFF.

```
op1 = unsigned32(swap(src1)) * 256;
op2 = unsigned32(swap(src2));
dst_lo = swap(0xFFFF);
dst_hi = swap(0xFFFF);
if (op2) {
  quo = op1 / op2;
  rem = op1 % op2;
  if (quo <= 0xFFFF) {
    dst_lo = swap(quo);
    dst_hi = swap(rem);
  }
}
```

### DIV32S, opcodeA = 01001110b, S = 0

Divides src1_hi:src1_lo by src2, treating both as signed integers. This is a 32-bit by 16-bit signed divide. DIV32S requires the S bit to be 0 in the opcode; opcodeA = 01001110b, S = 1 is reserved.

If src1 is an *extreg*, then src1_hi comes from *extreg* + 1, and src1_lo comes from *extreg*. If src1 is a native CPU register, then src1_hi comes from that register, and src1_lo is 0x0000.

Places the signed quotient in dst_lo and remainder in dst_hi. If src2 is 0 or the result is out of range, both quotient and remainder get 0x7FFF.

```
if (src2) {
  quo = signed32(src1_hi:src1_lo) / signed16(src2);
  rem = signed32(src1_hi:src1_lo) % signed16(src2);
}
if (!src2 || quo > 0x7FFF || quo < -0x8000) {
  quo = 0x7FFF;
  rem = 0x7FFF;
}
dst_lo = quo;
dst_hi = rem;
```

### DIV32U, opcodeA = 01001111b, S = 0

Divides src1_hi:src1_lo by src2, treating both as unsigned integers. This is a 32-bit by 16-bit unsigned divide. DIV32U requires the S bit to be 0 in the opcode; opcodeA = 01001111b, S = 1 is reserved.

If src1 is an *extreg*, then src1_hi comes from *extreg* + 1, and src1_lo comes from *extreg*. If src1 is a native CPU register, then src1_hi comes from that register, and src1_lo is 0x0000.

Places the unsigned quotient in dst_lo and remainder in dst_hi. If src2 is 0 or the result is out of range, both quotient and remainder get 0xFFFF.

```
if (src2) {
  quo = unsigned32(src1_hi:src1_lo) / unsigned16(src2);
  rem = unsigned32(src1_hi:src1_lo) % unsigned16(src2);
}
if (!src2 || quo > 0xFFFF) {
  quo = 0xFFFF;
  rem = 0xFFFF;
}
dst_lo = quo;
dst_hi = rem;
```

### ADDS, opcodeA = 01010000b, S = 0

Addition with sign-extension. Performs a 16-bit + 16-bit ⟹ 32-bit add, sign-extending both arguments to 32 bits first. This is intended for promoting 16-bit values to 32-bit, for certain forms of extended-precision arithmetic.

```
dst_hi:dst_lo = sign_extend32(src1) + sign_extend32(src2);
```

### ADDU, opcodeA = 01010000b, S = 1

Addition with zero-extension. Performs a 16-bit + 16-bit ⟹ 32-bit add, zero-extending both arguments to 32 bits first. This serves two purposes:

1. Promoting 16-bit values to 32-bit as part of extended-precision arithmetic, and
2. The first step of an N-bit + N-bit extended precision add, writing the carry in dst_hi.

For the second role, see the example under ADDM below.

```
dst_hi:dst_lo = zero_extend32(src1) + zero_extend32(src2);
```

### ADDH, opcodeA = 01010001b, S = 0

Adds the three 16-bit values in src1, src2, and dst, writing the result to dst. This is intended for two purposes:

1. An optimized three-input general purpose addition, and
2. The terminating step of an extended precision addition.

For the second role, see the example under ADDM below.

```
dst = src1 + src2 + dst;
```

### ADDM, opcodeA = 01010001b, S = 1

Adds the three 16-bit values in src1, src2, and dst_lo, writing the 32-bit result to dst_hi:dst_lo. This performs a 16-bit + 16-bit + 16-bit ⟹ 32-bit add, zero-extending input arguments to 32 bits first.

```
dst_hi:dst_lo = zero_extend32(src1) + zero_extend32(src2)
              + zero_extend(dst_lo);
```

### SUBS, opcodeA = 01010010b

Subtraction with sign-extension. Performs a 16-bit - 16-bit ⟹ 32-bit add, sign-extending both arguments to 32 bits first. This is intended for promoting 16-bit values to 32-bit, for certain forms of extended-precision arithmetic.

```
dst_hi:dst_lo = sign_extend32(src1) - sign_extend32(src2);
```

### SUBU, opcodeA = 01010011b

Subtraction with zero-extension. Performs a 16-bit - 16-bit ⟹ 32-bit add, zero-extending both arguments to 32 bits first. The result is effectively a 32-bit signed value. This serves two purposes:

3. Promoting 16-bit values to 32-bit as part of extended-precision arithmetic, and
4. The first step of an N-bit + N-bit extended precision subtract, writing the carry/borrow in dst_hi.

For the second role, see the example under ADDM above.

```
dst_hi:dst_lo = zero_extend32(src1) - zero_extend32(src2);
```

### SUBM, opcodeA = 01010100b

Adds the three 16-bit values in src1, src2, and dst_lo, writing the 32-bit result to dst_hi:dst_lo. This performs a 16-bit - 16-bit + 16-bit ⟹ 32-bit mixed add/subtract.

Both src1 and src2 are treated as unsigned, while dst_lo is treated as signed, to account for a negative 'borrow'. This is intended to serve as the middle step of an extended-precision subtract. See the example under ADDM above for additional details.

```
dst_hi:dst_lo = zero_extend32(src1) + zero_extend32(src2)
              + sign_extend(dst_lo);
```

### SUBH, opcodeA = 01010101b

Subtracts the 16-bit values in src2 from src1, and then adds the result to dst. This is intended to be the final step of an extended precision subtract. See the example under ADDM above for additional details.

```
dst = src1 - src2 + dst;
```

### DMOV, opcodeA = 01010110b

Double register move: Copies the values in src1 and src2 to dst_hi:dst_lo.

```
dst_hi = src1;
dst_lo = src2;
```

### ADDSUB, opcodeA = 01010111b

Computes the sum and difference between src1 and src2. Writes the sum to dst_hi, and the difference to dst_lo.

```
dst_hi = src1 + src2;
dst_lo = src1 - src2;
```

### ABCD, opcodeA = 01011000b, S = 0

Binary Coded Decimal (BCD) addition. Adds src1 and src2 as BCD inputs, writing the result to dst.

```
dst = src1 + src2;  // BCD
```

### ABCDL, opcodeA = 01011000b, S = 1

Binary Coded Decimal (BCD) addition with carry/borrow output. Adds src1 and src2, producing a 16-bit BCD result. Treats src1 and src2 as BCD inputs and generates the carry/borrow as described in BCD Formats.

Writes the result to dst_lo, and 3-bit signed carry/borrow to dst_hi. Intended for extended-precision BCD arithmetic. See example under ADDM above.

```
dst_hi:dst_lo = src1 + src2;  // BCD
```

### ABCDH, opcodeA = 01011001b, S = 0

Binary Coded Decimal (BCD) addition with carry/borrow input. Adds src1 and src2 along with a 3 bit carry/borrow input in dst_lo. Treats src1 and src2 as BCD inputs and interprets the carry/borrow as described in BCD Formats.

Writes the result to dst_lo. Intended for extended-precision BCD arithmetic. See example under ADDM above.

```
dst = src1 + src2 + bcd_carry_borrow(dst_lo);  // BCD
```

### ABCM, opcodeA = 01011001b, S = 1

Binary Coded Decimal (BCD) addition with carry/borrow output and carry/borrow input. Adds src1 and src2 along with a 3 bit carry/borrow input in dst_lo. Treats src1 and src2 as BCD inputs and interprets the carry/borrow as described in BCD Formats.

Writes the result to dst_lo, and 3-bit signed carry/borrow to dst_hi. Intended for extended-precision BCD arithmetic. See example under ADDM above.

```
dst_hi:dst_lo = src1 + src2 + bcd_carry_borrow(dst_lo);  // BCD
```

### SBCD, opcodeA = 01011010b

Binary Coded Decimal (BCD) subtract. Subtracts src2 from src1 as BCD inputs, writing the result to dst.

```
dst = src1 - src2;  // BCD
```

### SBCDL, opcodeA = 01011011b

Binary Coded Decimal (BCD) subtraction with carry/borrow output. Subtracts src2 from src1, producing a 16-bit BCD result. Treats src1 and src2 as BCD inputs and generates the carry/borrow as described in BCD Formats.

Writes the result to dst_lo, and 3-bit signed carry/borrow to dst_hi. Intended for extended-precision BCD arithmetic. See example under ADDM above.

```
dst_hi:dst_lo = src1 - src2;  // BCD
```

### SBCM, opcodeA = 01011100b

Binary Coded Decimal (BCD) subtraction with carry/borrow output and carry/borrow input. Subtracts src2 from src1 while adding a 3 bit carry/borrow input in dst_lo. Treats src1 and src2 as BCD inputs and interprets the carry/borrow as described in BCD Formats.

Writes the result to dst_lo, and 3-bit signed carry/borrow to dst_hi. Intended for extended-precision BCD arithmetic. See example under ADDM above.

```
dst_hi:dst_lo = src1 - src2 + bcd_carry_borrow(dst_lo);  // BCD
```

### SBCDH, opcodeA = 01011101b

Binary Coded Decimal (BCD) subtraction with carry/borrow input. Subtracts src2 from src1 while adding a 3 bit carry/borrow input in dst_lo. Treats src1 and src2 as BCD inputs and interprets the carry/borrow as described in BCD Formats.

Writes the result to dst_lo. Intended for extended-precision BCD arithmetic. See example under ADDM above.

```
dst = src1 - src2 + bcd_carry_borrow(dst_lo);  // BCD
```

### I2BCD, opcodeA = 01011110b

Converts the 32-bit unsigned integer value in src1:src2 to 32-bit BCD Format in dst_hi:dst_lo. Clamps the result to 0x99999999.

```
input = (unsigned32(src1) << 16) | unsigned32(src2);

if (input > 99999999) {
  result = 0x99999999;
} else {
  d0 = (input / 1) % 10;
  d1 = (input / 10) % 10;
  d2 = (input / 100) % 10;
  d3 = (input / 1000) % 10;
  d4 = (input / 10000) % 10;
  d5 = (input / 100000) % 10;
  d6 = (input / 1000000) % 10;
  d7 = (input / 10000000) % 10;
  result = (d7 << 28) | (d6 << 24) | (d5 << 20) | (d4 << 16)
         | (d3 << 12) | (d2 <<  8) | (s1 <<  4) | (d0 <<  0);
}

dst_hi:dst_lo = result;
```

### BCD2I, opcodeA = 01011111b

Converts the 32-bit BCD format input in src1:src2 into a 32-bit unsigned integer in dst_hi:dst_lo.

```
input = (src1 << 16) | src2;

d7 = (input >> 28) & 0xF;
d6 = (input >> 24) & 0xF;
d5 = (input >> 20) & 0xF;
d4 = (input >> 16) & 0xF;
d3 = (input >> 12) & 0xF;
d2 = (input >>  8) & 0xF;
d1 = (input >>  4) & 0xF;
d0 = (input >>  0) & 0xF;

result = d7 * 10000000
       + d6 * 1000000
       + d5 * 100000
       + d4 * 10000
       + d3 * 1000
       + d2 * 100
       + d1 * 10
       + d0;

dst_hi:dst_lo = result;
```

### CMPEQ&, opcodeA = 01100000b, S = 0

Bitwise-ANDs dst with 0 or -1 (0xFFFF) based on whether src1 equals src2.

```
dst &= src1 == src2 ? -1 : 0;
```

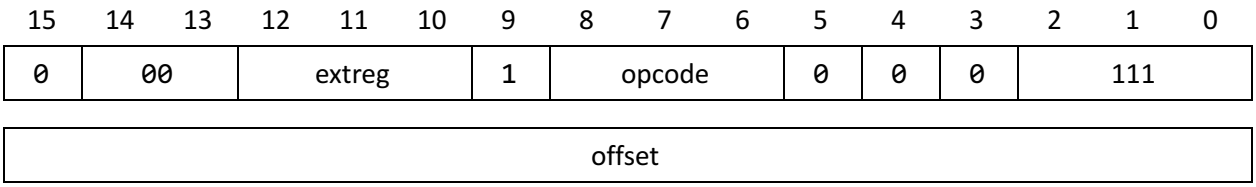
### CMPNE&, opcodeA = 01100000b, S = 1

Bitwise-ANDs dst with 0 or -1 (0xFFFF) based on whether src1 equals src2.

```
dst &= src1 != src2 ? -1 : 0;
```

## Extended Conditional Branches

CP-1600X supports a small number of extended conditional branches. These are built from the extended addressing mode support. CP-1600X treats amode=00b, reg=111b as an extended conditional branch. Thus, the opcode encoding looks as follows:




Opcode Field Definitions

| Field | Meaning |
|---|---|
| extreg | External register X1 through X7. (001b = X1, 111b = X7, etc.) |
| opcode | CP-1600 instruction opcode |
| offset | Address offset to apply in extended addressing modes (amode ≠ 0 or extreg ≠ 0) |

These branches overload the meaning of the CP-1600's arithmetic operations that target R7. That limits the number of potential extended branches. Currently, CP-1600X defines behavior for the following opcodes:

| Opcode | Original Mnemonic | Extended ISA Mnemonic | Description |
|---|---|---|---|
| 001 | MVO | - | No change / reserved. |
| 010 | MVI | TSTBNZ | Test Xreg and branch if non-zero. |
| 011 | ADD | TXSER / RXSER | Specialized serial transmit/receive with branch. |
| 100 | SUB | - | No change / reserved. |
| 101 | CMP | - | No change / reserved. |
| 110 | AND | - | No change / reserved. |
| 111 | XOR | DECBNZ | Decrement Xreg and branch if non-zero. |

The TSTBNZ and DECBNZ opcodes offer a single 16-bit branch destination. The TXSER/RXSER offer *two* branch destinations, to cover "data available" and "serial error" cases.

## TSTBNZ / DECBNZ

The TSTBNZ instruction provides a “test, and branch if zero” instruction. It takes a single extreg and branch target as arguments. If the extreg is non-zero, it branches to the target address. This instruction does not modify the CP-1600 flags.

The DECBNZ instruction operates similarly; however, it decrements the extreg before testing it. Because DECBNZ is built from XOR, it *will* modify the CP-1600 Sign and Zero flags based the address the CPU arrives at.

The offset in the second word of the instruction is indeed an offset, and not an absolute address.

It’s not clear at the time of writing whether these instructions take 10 or 11 cycles. (Most likely, 11 cycles). This is still faster than a typical, native test/branch or decrement/branch (15 cycles).

## TXSER / RXSER

These instructions are intended to speed up serial I/O when used with an actual Locutus cartridge. They are not supported in jzIntv, and may not be supported in a JLP-style setting.

The instructions have the following syntax:

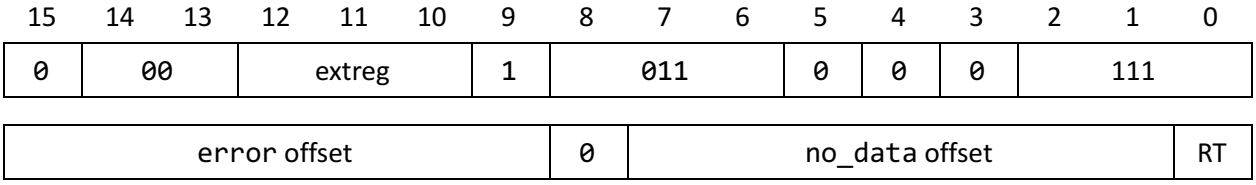
```
TXSER    extreg, no_data, error
RXSER    extreg, no_data, error
```

Here, *extreg* is one of X1 through X7. *no_data* and *error* are labels.

These instructions combine data transfer between the serial port and an extreg with a three-way branch. The TXSER / RXSER instructions always do one of the following:

1. Transfer data between serial port and extreg and fall through tothe next instruction, or
2. Branch to the no_data label if no data is available, or
3. Branch to the error label if a serial error occurred.

The two branch offsets, as well as RXSER vs. TXSER are encoded into the offset field as follows. RT = 0 means RXSER; RT = 1 means TXSER.



# Programmer's Guide

This section provides additional insight on how the CP-1600X instructions support various uses.

## Extended Precision Addition and Subtraction

### Extended Precision Integer Addition

The ADDU, ADDM, and ADDH instructions work together to provide an extended-precision addition facility. The ADDU instruction starts the addition, writing a carry its dst_hi. The ADDM instruction consumes the carry, and produces a new carry in its dst_hi. The ADDH instruction terminates the addition, consuming the final carry but producing none of its own.

The following example adds the 64-bit value in X3:X2:X1:X0 to X7:X6:X5:X4, placing the result in XB:XA:X9:X8.

```
        ADDU  X0,  X4,  X8  ; writes to X9:X8
        ADDM  X1,  X5,  X9  ; consumes carry in X9, writes to XA:X9
        ADDM  X2,  X6,  XA  ; consumes carry in XA, writes to XB:XA
        ADDH  X3,  X7,  XB  ; consumes carry in XB, writes to XB
```

One downside of this design is that you cannot really perform "in-place" accumulation, as the carry flag always occupies the adjacent register to the sum. Two DMOV instructions can clean this up:

```
        DMOV  X9,  X8,  X4  ; copies X9:X8 to X5:X4
        DMOV  XB,  XA,  X6  ; copies XB:XA to X7:X6
```

*Word of caution:* As each of these instructions is non-interruptible, you may need a NOP or other interruptible instruction somewhere in the sequence for a 64-bit or higher precision addition.

A 32-bit extended precision addition only requires ADDU and ADDH, and at most one DMOV if you wish to retain the 32-bit value in its starting registers:

```
; Add X1:X0 + X3:X2 => X3:X2, using X5:X4 as temporaries
        ADDU  X0,  X2,  X4  ; writes to X5:X4
        ADDH  X1,  X3,  X5  ; consumes carry in X5; writes to X5
        DMOV  X5,  X4,  X2  ; copies X5:X4 to X3:X2
```

These instructions are mainly useful for extended precision values that live for extended periods of time in extregs. For values that primarily live in RAM, the native CPU instructions with ADD@/ADCR may be a

better choice, as CP-1600X does not yet have an efficient mechanism to MVI@ or MVO@ to/from an extreg.

## Extended Precision Integer Subtraction

Extended precision subtraction follows the same pattern as extended precision addition. Rather than writing a carry result to the second register, however, the SUBU and SUBM instructions write a *borrow* instead. That is, the output is 0 if there was no borrow, or -1 (0xFFFF) if there was a borrow.

The following example subtracts the 64-bit value in X3:X2:X1:X0 from X7:X6:X5:X4, placing the result in XB:XA:X9:X8.

```
        SUBU  X4,  X0,  X8  ; writes to X9:X8
        SUBM  X5,  X1,  X9  ; consumes borrow in X9, writes to XA:X9
        SUBM  X6,  X2,  XA  ; consumes borrow in XA, writes to XB:XA
        SUBH  X7,  X3,  XB  ; consumes borrow in XB, writes to XB
```

The equivalent 32-bit example, subtracting X1:X0 from X3:X2, writing to X5:X4:

```
        SUBU  X2,  X0,  X4  ; writes to X5:X4
        SUBH  X3,  X1,  X5  ; consumes carry in X5; writes to X5
```

## Extended Precision BCD Addition and Subtraction

The BCD addition and subtraction instructions follow the same pattern as the integer versions. Because the BCD instructions define their carry/borrow differently, and because the BCD representation is inherently unsigned, the mnemonic for the first instruction of each sequence ends in an 'L' rather than a 'U'.

ABCD stands for "Add Binary Coded Decimal", while SBCD stands for "Subtract Binary Coded Decimal". Therefore, these examples look very similar to the ADD/SUB examples above, with ADD/SUB replaced by ABCD/SBCD, and the 'U' rewritten to 'L'.

Adding the 16-digit (64-bit) BCD value in X3:X2:X1:X0 to X7:X6:X5:X4, placing the result in XB:XA:X9:X8, looks as follows:

```
        ABCDU X0,  X4,  X8  ; writes to X9:X8
        ABCDM X1,  X5,  X9  ; consumes carry in X9, writes to XA:X9
        ABCDM X2,  X6,  XA  ; consumes carry in XA, writes to XB:XA
        ABCDH X3,  X7,  XB  ; consumes carry in XB, writes to XB
```

Adding the 8-digit (32-bit) BDC value in X1:X0 to X3:X2 with the result in X5:X4 looks as follows:

```
        ABCDU X0,  X2,  X4  ; writes to X5:X4
        ABCDH X1,  X3,  X5  ; consumes carry in X5, writes to X5
```

Subtracting the 16-digit (64-bit) BCD value in X3:X2:X1:X0 from X7:X6:X5:X4, placing the result in XB:XA:X9:X8, looks as follows:

```
        SBCDU X4,  X0,  X8  ; writes to X9:X8
        SBCDM X5,  X1,  X9  ; consumes borrow in X9, writes to XA:X9
        SBCDM X6,  X2,  XA  ; consumes borrow in XA, writes to XB:XA
        SBCDH X7,  X3,  XB  ; consumes borrow in XB, writes to XB
```

Subtracting the 8-digit (32-bit) BDC value in X1:X0 from X3:X2 with the result in X5:X4 looks as follows:

```
        SBCDU X0,  X2,  X4  ; writes to X5:X4
        SBCDH X1,  X3,  X5  ; consumes borrow in X5, writes to X5
```

# Revision History

| Date | Notes |
|---|---|
| 17-Nov-2019, A | Initial, partially complete release. Most, but not all, instructions described. |
| 17-Nov-2019, B | Fix a couple minor errors, typos. |
| 19-Nov-2019, A | Remaining 3-op extended ISA, description of BCD operand types, and minor fixes; still to-do: TSTBNZ, DECBNZ, TXSER, RXSER, and full description of BCD arithmetic. |
| 3-Dec-2019, A | Added short description of PV and register pairs; added diagram to ATAN2; started “Programmer’s Guide” section and migrated extended precision, BCD documentation there; fixed a number of instruction descriptions and mnemonics to align with macro file. |