PRU Assembly Instructions

^ Up to main **Programmable Realtime Unit and Industrial Communication Subsystem (PRU-ICSS)** Table of Contents

This article is part of a collection of articles describing software development on the PRU subsystem included in OMAP-L1x8/C674m/AM18xx/AM335x/AM437x/AM57xx/66AK2Gx devices (check the device data sheet for PRU/PRU-ICSS support). To navigate to the main PRU software development page click on the link above.

PRU Core Revision

There are two main PRU Core Revisions that have been implemented on TI devices. The table below summarizes the difference between the supported assembly set for each revision. Note that even though some of these functions are supported by a particular core revision, there may be additional hardware dependencies that are not implemented on a given device.

In general, core revision 1 has the largest common instruction set, and thus when uncertain about the target core or when binary support for multiple core revisions is needed, assemble with the –V1 option. Code written for a revision 1 core can execute on later cores by avoiding the SCAN instruction, but assembling for later cores will increase the efficiency of some instructions.

PRU Core Revision Comparison

Assembler Instruction	V1	V3
LFC / STC	-	-
SCAN	Yes	-
MVI	Pseudo Op (limited)	Yes
SLP	Yes (adds trailing NOP)	Yes
ZERO	Pseudo Op (multi-cycle)	Yes
Fill	-	Yes
XIN/XOUT	-	Yes
LOOP /ILOOP	-	Yes
NOP*n*	-	Yes

pasm vs. clpru

There are two assemblers available for PRU, pasm and clpru. The original assembler for PRU was pasm. This assembler supports a single translation unit and assembles directly to a binary image (or other compatible format). The clpru tool is actually a full C/C++ compiler toolchain for PRU and includes an assembler. The assembler included in the clpru toolchain supports multiple translation units and assembles to object files which must be linked together into a final executable. The two assemblers support nearly the same instruction syntax. The differences are listed in the description of each instruction. The non-instruction syntax is quite different and includes things like symbol defintion, macros, etc.

Key differences

- **Input file extension**

  The pasm assembler uses '.p' whereas the clpru assembler uses '.asm'

- **Comments**

  The pasm assembler uses // and /* */ for comments. The clpru assembler uses ';' to comment out the remaining text on a line.

- **Symbol value macros.**

  The pasm assembler uses C preprocessor style macros:

```
#define SYM1 SYM2
#define SYM1 10
```

  The clpru assembler uses the .set or .asg directives:

```
SYM1 .set SYM2
SYM1 .set 10
```

```
   .asg "10",      SYM1
   .asg "SYM1",    SYM2
   .asg "r24.w0",  SYM3
```

The .set directive defines a new symbol, which will be included in the resulting object file. The .asg directive creates a substitution symbol and is handled by string substitution. The .asg directive is likely the right directive to replace #define. *** Conditional compilation** The pasm assembler uses the C preprocessor syntax of #if, #else, #endif:

```
#ifdef <MACRO>
...
#endif
```

The clpru assembler uses the .if, .else, and .endif directives:

```
.if $isdefed("<MACRO>")
...
.endif
```

- **Register modifier suffix**

  The pasm assembler allows either .w0 or .W0, whereas the clpru assembler requires register modifiers to be lower case.

- **Instruction differences**
  - MOV REG1, IM is only supported in pasm. Use LDI in clpru.
  - The clpru assembler requires the source/destination register in LBCO, SBCO, LBBO, and SBBO instructions to be prefixed with '&':

```
 LBBO &R0, R1, 10
```

  - The clpru assembler does not accept a single operand for the source and destination of SET/CLR.

```
 SET Rx, Rx.ty
```

NOT

```
 SET Rx.ty
```

- The clpru assembler requires '&' on the first operand fo XIN/XOUT
- LFC, SFC, and SCAN are not supported in clpru.
- The CALL, RET, and .setcallreg directive are not support in clpru. Instead you must use the JAL and JMP instruction directly.

- **Loading symbol addresses into registers**

  The pasm assembler only supports symbols for code addresses and therefore always uses LDI/MOV R0, <sym>. Furthermore LDI R0, <sym> will always load the word address of <sym>. The clpru assembler supports symbols for both code and data. Code addresses are always 16-bits and word addresses, but data addresses are either 16-bit or 32-bit and byte addresses. The following idoms are used in the clpru assembler:

  16-bit data address:

```
LDI R0, <sym>
```

  32-bit data address:

```
LDI32 R0, <sym>
```

  16-bit code address:

```
LDI R0, $CODE(<sym>)
```

- **Macros**

  The pasm assembler defines macros with the macro name after the .macro directive:

```
.macro <MACRO>
```

  The clpru assembler defines macros with the macro name before the .macro directive:

```
<MACRO> .macro
```

For more information on the capabilities of the clpru assembler please refer to PRU Assembly Language Tools User Guide [1].

Instruction Set Syntax Terminology

Table 1 provides terminology needed to understand the syntax for the instruction set.

Table 1: Instruction Set syntax Terminology

Parameter Name	Meaning	Examples
REG, REG1, REG2, ...	Any register field from 8 to 32 bits	r0, r1.w0, r3.b2
Rn, Rn1, Rn2, ...	Any 32 bit register field (r0 through r31)	r0, r1
Rn.tx	Any 1 bit register field (x dentoes the bit position)	r0.t23, r1.b2.t5
Rn.bx	Specifies a byte field that must be b0, b1, b2, or b3 – denoting r0.b0, r0.b1, r0.b2, and r0.b3 respectively.	b0, b1
Rn.wx	Specifies a two byte (word) field that must be w0, w1, or w2 - denoting r0.w0, r0.w1, and r0.w2 respectively. w0 spans bytes 0 and 1; w1 spans bytes 1 and 2; w2 spans bytes 2 and 3.	w0, w1
Cn, Cn1, Cn2, ...	Any 32 bit constant table entry (c0 through c31)	c0,c1

LABEL	Any valid label, specified with or without parenthesis. An immediate value denoting an instruction address is also acceptable.	loop1, (loop1), 0x0000
IM(n)	An immediate value from 0 to n. In clpru immediate values should be specified without a leading hash \"\#\". In pasm, the leading \"\#\" is accepted, but optional. Immediate values, labels, and register addresses are all acceptable.	#23, 0b0110, 0xF2, 2+2, &r3.w2
OP(n)	The union of REG and IM(n)	r0, r1.w0, #0x7F, 1<<3, loop1, &r1.w0

For example the following is the definition for the ADD instruction:

```
ADD REG1, REG2, OP(255)
```

This means that the first and second parameters can be any register field from 8 to 32 bits. The third parameter can be any register field from 8 to 32 bits or an immediate value from 0 to 255. Thus the following are all legal ADD instructions:

```
ADD R1, R1, #0x25 // r1 += 37
ADD r1, r1, 0x25 // r1 += 37
ADD r3, r1, r2 // r3 = r1 + r2
ADD r1.b0, r1.b0, 0b100 // r1.b0 += 4
ADD r2, r1.w0, 1<<3 // r2 = r1.w0 + 8
```

Instruction set

Arithmetic and Logical

All operations are 32 bits wide (with a 33 bit result in the case of arithmetic's). The source values are zero extended prior to the operation. If the destination is too small to accept the result, the result is truncated.

On arithmetic operations, the first bit to the left of the destination width becomes the carry value. Thus if the destination register is an 8 bit field, bit 8 of the result becomes the carry. For 16 and 32 bit destinations, bit 16 and bit 32 are used as the carry bit respectively.

Unsigned Integer Add (ADD)

Performs 32-bit add on two 32 bit zero extended source values.

Syntax:

```
ADD REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 + OP(255)
carry = (( REG2 + OP(255) ) >> bitwidth(REG1)) & 1
```

Example:

```
 add    r3, r1, r2
 add    r3, r1.b0, r2.w2
 add    r3, r3, 10
```

Unsigned Integer Add with Carry (ADC)

Performs 32-bit add on two 32 bit zero extended source values, plus a stored carry bit.

Definition:

```
ADC REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 + OP(255) + carry
carry = (( REG2 + OP(255) + carry ) >> bitwidth(REG1)) & 1
```

Example:

```
 adc     r3, r1, r2
 adc     r3, r1.b0, r2.w2
 adc     r3, r3, 10
```

Unsigned Integer Subtract (SUB)

Performs 32-bit subtract on two 32 bit zero extended source values

Definition:

```
SUB REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 - OP(255)
carry = (( REG2 - OP(255) ) >> bitwidth(REG1)) & 1
```

Example:

```
 sub     r3, r1, r2
 sub     r3, r1.b0, r2.w2
 sub     r3, r3, 10
```

Unsigned Integer Subtract with Carry (SUC)

Performs 32-bit subtract on two 32 bit zero extended source values with carry (borrow)

Definition:

```
SUC REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 - OP(255) - carry
carry = (( REG2 - OP(255) - carry ) >> bitwidth(REG1)) & 1
```

Example:

```
 suc     r3, r1, r2
 suc     r3, r1.b0, r2.w2
 suc     r3, r3, 10
```

Reverse Unsigned Integer Subtract (RSB)

Performs 32-bit subtract on two 32 bit zero extended source values. Source values reversed.

Definition:

```
RSB REG1, REG2, OP(255)
```

Operation:

```
REG1 = OP(255) - REG2
carry = (( OP(255) - REG2 ) >> bitwidth(REG1)) & 1
```

Example:

```
 rsb     r3, r1, r2
 rsb     r3, r1.b0, r2.w2
 rsb     r3, r3, 10
```

Reverse Unsigned Integer Subtract with Carry (RSC)

Performs 32-bit subtract on two 32 bit zero extended source values with carry (borrow). Source values reversed.

Definition:

```
RSC REG1, REG2, OP(255)
```

Operation:

```
REG1 = OP(255) - REG2 - carry
carry   = (( OP(255) - REG2 - carry ) >> bitwidth(REG1)) & 1
```

Example:

```
 rsc     r3, r1, r2
 rsc     r3, r1.b0, r2.w2
 rsc     r3, r3, 10
```

Logical Shift Left (LSL)

Performs 32-bit shift left of the zero extended source value

Definition:

```
LSL REG1, REG2, OP(31)
```

Operation:

```
REG1 = REG2 << ( OP(31) & 0x1f )
```

Example:

```
 lsl     r3, r3, 2
 lsl     r3, r3, r1.b0
 lsl     r3, r3.b0, 10
```

Logical Shift Right (LSR)

Performs 32-bit shift right of the zero extended source value

Definition:

```
LSR REG1, REG2, OP(31)
```

Operation:

```
REG1 = REG2 >> ( OP(31) & 0x1f )
```

Example:

```
 lsr     r3, r3, 2
 lsr     r3, r3, r1.b0
 lsr     r3, r3.b0, 10
```

Bitwise AND (AND)

Performs 32-bit logical AND on two 32 bit zero extended source values.

Definition:

```
AND REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 & OP(255)
```

Example:

```
 and     r3, r1, r2
 and     r3, r1.b0, r2.w2
 and     r3.b0, r3.b0, ~(1<<3)    // Clear bit 3
```

Bitwise OR (OR)

Performs 32-bit logical OR on two 32 bit zero extended source values.

Definition:

```
OR REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 | OP(255)
```

Example:

```
 or      r3, r1, r2
 or      r3, r1.b0, r2.w2
 or      r3.b0, r3.b0, 1<<3     // Set bit 3
```

Bitwise Exclusive OR (XOR)

Performs 32-bit logical XOR on two 32 bit zero extended source values.

Definition:

```
XOR REG1, REG2, OP(255)
```

Operation:

```
REG1 = REG2 ^ OP(255)
```

Example:

```
 xor      r3, r1, r2
 xor      r3, r1.b0, r2.w2
 xor      r3.b0, r3.b0, 1<<3   // Toggle bit 3
```

Bitwise NOT (NOT)

Performs 32-bit logical NOT on the 32 bit zero extended source value.

Definition:

```
NOT REG1, REG2
```

Operation:

```
REG1 = ~REG2
```

Example:

```
 not      r3, r3
 not      r1.w0, r1.b0
```

Copy Minimum (MIN)

Compares two 32 bit zero extended source values and copies the minimum value to the destination register.

Definition:

```
MIN REG1, REG2, OP(255)
```

Operation:

```
if( OP(255) > REG2 )
    REG1 = REG2;
else
    REG1 = OP(255);
```

Example:

```
 min      r3, r1, r2
 min      r1.w2, r1.b0, 127
```

Copy Maximum (MAX)

Compares two 32 bit zero extended source values and copies the maximum value to the destination register.

Definition:

```
MAX REG1, REG2, OP(255)
```

Operation:

```
if( OP(255) > REG2 )
    REG1 = OP(255);
else
    REG1 = REG2;
```

Example:

```
 max      r3, r1, r2
 max      r1.w2, r1.b0, 127
```

Clear Bit (CLR)

Clears the specified bit in the source and copies the result to the destination. Various calling formats are supported:

Format 1

Definition:

```
CLR REG1, REG2, OP(31)
```

Operation:

```
REG1 = REG2 & ~( 1 << (OP(31) & 0x1f) )
```

Example:

```
 clr      r3, r1, r2          // r3 = r1 & ~(1<<r2)
 clr      r1.b1, r1.b0, 5     // r1.b1 = r1.b0 & ~(1<<5)
```

Format 2 (same source and destination)

NOTE: This format is only supported in the pasm assembler.

Definition:

```
CLR REG1, OP(31)
```

Operation:

```
REG1 = REG1 & ~( 1 << (OP(31) & 0x1f) )
```

Example:

```
 clr      r3, r1             // r3 = r3 & ~(1<<r1)
 clr      r1.b1, 5           // r1.b1 = r1.b1 & ~(1<<5)
```

Format 3 (source abbreviated)

Definition:

```
CLR REG1, Rn.tx
```

Operation:

```
REG1 = Rn & ~(1<<x)
```

Example:

```
 clr      r3, r1.t2          // r3 = r1 & ~(1<<2)
 clr      r1.b1, r1.b0.t5    // r1.b1 = r1.b0 & ~(1<<5)
```

Format 4 (same source and destination – abbreviated)

NOTE: This format is only supported in the pasm assembler.

Definition:

```
CLR Rn.tx
```

Operation:

```
Rn = Rn & ~(1<<x)
```

Example:

```
 clr      r3.t2              // r3 = r3 & ~(1<<2)
```

Set Bit (SET)

Sets the specified bit in the source and copies the result to the destination. Various calling formats are supported.

NOTE: Whenever R31 is selected as the source operand to a SET, the resulting source bits will be NULL, and not reflect the current input event flags that are normally obtained by reading R31.

Format 1

Definition:

```
SET REG1, REG2, OP(31)
```

Operation:

```
REG1 = REG2 | ( 1 << (OP(31) & 0x1f) )
```

Example:

```
 set      r3, r1, r2          // r3 = r1 | (1<<r2)
 set      r1.b1, r1.b0, 5     // r1.b1 = r1.b0 | (1<<5)
```

Format 2 (same source and destination)

NOTE: This format is only supported in the pasm assembler.

Definition:

```
SET REG1, OP(31)
```

Operation:

```
REG1 = REG1 | ( 1 << (OP(31) & 0x1f) )
```

Example:

```
 set      r3, r1              // r3 = r3 | (1<<r1)
 set      r1.b1, 5            // r1.b1 = r1.b1 | 1<<5)
```

Format 3 (source abbreviated)

Definition:

```
SET REG1, Rn.tx
```

Operation:

```
REG1 = Rn | (1<<x)
```

Example:

```
 set      r3, r1.t2          // r3 = r1 | (1<<2)
 set      r1.b1, r1.b0.t5    // r1.b1 = r1.b0 | (1<<5)
```

Format 4 (same source and destination – abbreviated)

NOTE: This format is only supported in the pasm assembler.

Definition:

```
SET Rn.tx
```

Operation:

```
Rn = Rn | (1<<x)
```

Example:

```
 set      r3.t2              // r3 = r3 | (1<<2)
```

Register Field Scan (SCAN)

NOTE: This instruction is only supported in the pasm assembler.

The SCAN instruction scans the register file for a particular value. It includes a configurable field width and stride. The width of the field to match can be set to 1, 2, or 4 bytes. The span between fields in bytes is programmable from 1 to 4 bytes. (Having a stride independent of width allows the programmer to scan for non-byte values on byte boundaries. For example, scan for "7F03" on a byte by byte basis). *This instruction is deprecated and not available on all PRU cores.*

Definition:

```
SCAN Rn, OP(255)
```

Operation:

```
The register "Rn" serves as both the source and results register. It is coded as follows:
```

Rn.b0	Byte offset from the start of the register file to begin the scan (see section 3.3 for details on register addressing)
Rn.b1	Number of fields to scan
Rn.b2	Byte width of field to scan for (1, 2, 4)
Rn.b3	Byte stride of consecutive fields (1 to 4)

The instruction scans for the value specified in OP(255). On completion, the Rn register holds the results of the scan. It is coded as follows:

Rn.b0	Byte offset from R0.b0 to the matching field (or 0xFF if no match)
Rn.b1	Number of fields left to scan (including the matched field if any)
Rn.b2	*Not altered*
Rn.b3	*Not altered*

To continue a scan after a match, the programmer can write:

```
ADD     R1.w0, R1.w0, #0xFF01 // Inc byte offset, dec count
SCAN    R1, OP(255)
```

Example: Scan the register file for the sequence "0x7F 0x03" starting at R2.b1 and extending for 18 bytes. Do not assume the sequence is word aligned.

```
LDI     R1.w0, 0x7F | 0x03<<8  // 0x7F 0x03 in little endian
LDI     R30.w2, 2 | 1<<8       // Field width of 2, stride of 1
LDI     R30.w0, &r2.b1 | 18<<8 // Start at R2.b1, scan 18 bytes
SCAN    R30, R1.w0             // Scan for byte sequence
QBEQ    NOT_FOUND, R30.b1, 0   // Jump if sequence not found
```

Left-Most Bit Detect (LMBD)

Scans REG2 from its left-most bit for a bit value matching bit 0 of OP(255), and writes the bit number in REG1 (writes 32 to REG1 if the bit is not found).

Definition:

```
LMBD REG1, REG2, OP(255)
```

Operation:

```
for( i=(bitwidth(REG2)-1); i>=0; i-- )
if( !(( REG2>>i) ^ OP(255))&1) )
    break;
if( i<0 )
    REG1 = 32;
else
    REG1 = i;
```

Example:

```
lmbd     r3, r1, r2
lmbd     r3, r1, 1
lmbd     r3.b3, r3.w0, 0
```

Register Load and Store

Copy Value (MOV)

The MOV instruction moves the value from REG2, zero extends it, and stores it into REG1. The instruction is a pseudo op, and is coded with the instruction AND REG1, REG2, REG2. To load an immediate value into a register, see the LDI instruction.

Definition:

```
MOV REG1, REG2
```

Operation:

```
REG1 = REG2
```

Example:

```
 mov     r3, r1
 mov     r3, r1.b0        // Zero extend r1.b0 into r3
```

The pasm assembler supports MOV REG1, OP(65535). Examples of this form are:

```
 mov     r1, 10           // Move 10 into r1
 mov     r1, #10          // Move 10 into r1
 mov     r1, 0b10 + 020/2 // Move 10 into r1
 mov     r30.b0, &r2      // Move the offset of r2 into r30.b0
```

Load Immediate (LDI)

The LDI instruction moves the value from IM(65535), zero extends it, and stores it into REG1.

Definition:

```
LDI REG1, IM(65535)
```

Operation:

```
REG1 = IM(65535)
```

Example:

```
 ldi     r1, 10           // Load 10 into r1
 ldi     r1, 0b10 + 020/2 // Load 10 into r1
 ldi     r30.b0, &r2      // Load the offset of r2 into r30.b0
```

Move Register File Indirect (MVIx)

The MVIx instruction family moves an 8 bit, 16 bit, or 32 bit value from the source to the destination. The size of the value is determined by the exact instruction used; MVIB, MVIW, and MVID, for 8 bit, 16 bit, and 32 bit values respectively. The source, destination, or both must be a register pointer. There is an option for auto-increment and auto-decrement on register pointers. These instructions are only supported for core revisions V2 and later.

Definition:

```
 MVIB      [*][--]REG1[++], [*][--]REG2[++]
 MVIW      [*][--]REG1[++], [*][--]REG2[++]
 MVID      [*][--]REG1[++], [*][--]REG2[++]
```

Operation:

- Either the source or destination must be a register pointer restricted to r1.b0, r1.b1, r1.b2, or r1.b3
- Register pointers are byte offsets into the register file
- Auto increment and decrement operations are done by the byte width of the operation
  - Increments are post-increment; incremented after the register offset is used
  - Decrements are pre-decrement; decremented before the register offset is used
- When the source or destination registers are not expressed as register pointers, the size of the data read or written is determined by the field width of the register. If the data transfer size is less than the width of the destination, the data is zero extended. Size conversion occurs after indirect reads, and before indirect writes.

Load Byte Burst (LBBO)

The LBBO instruction is used to read a block of data from memory into the register file. The memory address to read from is specified by a 32 bit register (Rn2), using an optional offset. The destination in the register file can be specified as a direct register, or indirectly through a register pointer.

NOTE: In the pasm assembler either the traditional direct register syntax or the more recent register address offset syntax can be used for the first parameter.

Format 1 (immediate count)

Definition:

```
LBBO &REG1, Rn2, OP(255), IM(124)
```

Operation:

```
memcpy( offset(REG1), Rn2+OP(255), IM(124) );
```

Example:

```
 lbbo     &r2, r1, 5, 8     // Copy 8 bytes into r2/r3 from the
                            // memory address r1+5
```

Format 2 (register count)

Definition:

```
LBBO &REG1, Rn2, OP(255), bn
```

Operation:

```
memcpy( offset(REG1), Rn2+OP(255), bn );
```

Example:

```
 lbbo     &r3, r1, r2.w0, b0 // Copy "r0.b0" bytes into r3 from the
                             // memory address r1+r2.w0
```

Important Usage Note

For Format 2, do not use a byte count of 0 provided in R0.bn. It could cause the PRU to hang.

Store Byte Burst (SBBO)

The SBBO instruction is used to write a block of data from the register file into memory. The memory address to write to is specified by a 32 bit register (Rn2), using an optional offset. The source in the register file can be specified as a direct register, or indirectly through a register pointer.

NOTE: In the pasm assembler, either the traditional direct register syntax or the more recent register address offset syntax can be used for the first parameter.

Format 1 (immediate count)

Definition:

```
SBBO &REG1, Rn2, OP(255), IM(124)
```

Operation:

```
memcpy( Rn2+OP(255), offset(REG1), IM(124) );
```

Example:

```
 sbbo     &r2, r1, 5, 8     // Copy 8 bytes from r2/r3 to the
                            // memory address r1+5
```

Format 2 (register count)

Definition:

```
SBBO &REG1, Rn2, OP(255), bn
```

Operation:

```
memcpy( Rn2+OP(255), offset(REG1), bn );
```

Example:

```
 sbbo     &r3, r1, r2.w0, b0 // Copy "r0.b0" bytes from r3 to the
                             // memory address r1+r2.w0
```

Important Usage Note

For Format 2, do not use a byte count of 0 provided in R0.bn. It could cause the PRU to hang.

Load Byte Burst with Constant Table Offset (LBCO)

The LBCO instruction is used to read a block of data from memory into the register file. The memory address to read from is specified by a 32 bit constant register (Cn2), using an optional offset from an immediate or register value. The destination in the register file is specified as a direct register.

NOTE: In the pasm assembler, either the traditional direct register syntax or the more recent register address offset syntax can be used for the first parameter.

Format 1 (immediate count)

Definition:

```
LBCO &REG1, Cn2, OP(255), IM(124)
```

Operation:

```
memcpy( offset(REG1), Cn2+OP(255), IM(124) );
```

Example:

```
 lbco     &r2, c1, 5, 8     // Copy 8 bytes into r2/r3 from the
                            // memory address c1+5
```

Format 2 (register count)

Definition:

```
LBCO &REG1, Cn2, OP(255), bn
```

Operation:

```
memcpy( offset(REG1), Cn2+OP(255), bn );
```

Example:

```
 lbco     &r3, c1, r2.w0, b0 // Copy "r0.b0" bytes into r3 from the
                             // memory address c1+r2.w0
```

Important Usage Note

For Format 2, do not use a byte count of 0 provided in R0.bn. It could cause the PRU to hang.

Store Byte Burst with Constant Table Offset (SBCO)

The SBCO instruction is used to write a block of data from the register file into memory. The memory address to write to is specified by a 32 bit constant register (Cn2), using an optional offset from an immediate or register value. The source in the register file is specified as a direct register.

NOTE: In the pasm assembler either the traditional direct register syntax or the more recent register address offset syntax can be used for the first parameter.

Format 1 (immediate count)

Definition:

```
SBCO &REG1, Cn2, OP(255), IM(124)
```

Operation:

```
memcpy( Cn2+OP(255), offset(REG1), IM(124) );
```

Example:

```
 sbco     &r2, c1, 5, 8     // Copy 8 bytes from r2/r3 to the
                            // memory address c1+5
```

Format 2 (register count)

Definition:

```
SBCO &REG1, Cn2, OP(255), bn
```

Operation:

```
memcpy( Cn2+OP(255), offset(REG1), bn );
```

Example:

```
 sbco     &r3, c1, r2.w0, b0 // Copy "r0.b0" bytes from r3 to the
                             // memory address c1+r2.w0
```

Important Usage Note

For Format 2, do not use a byte count of 0 provided in R0.bn. It could cause the PRU to hang.

Load from Coprocessor (LFC)

The LFC instruction is used to load data from an external "coprocessor register" using an 8 bit coprocessor register address. *This instruction is deprecated and not available on all PRU cores.*

This instruction is not supported in the clpru assembler.

Definition:

```
LFC REG1, IM(255)
```

Operation:

```
REG1 = Coprocessor Register IM(255)
```

Example:

```
lfc     r2, 5      // Read coprocessor register 5
```

Store to Coprocessor (STC)

The STC instruction is used to write data to an external "coprocessor register" using an 8 bit coprocessor register address. Optionally, bits 31:24 can be specified from a third parameter OP(255) . *This instruction is deprecated and not available on all PRU cores.*

This instruction is not supported in the clpru assembler.

Format 1

Definition:

```
STC REG1, IM(255)
```

Operation:

```
Coprocessor Register IM(255) = (unsigned32) REG1
```

Example:

```
stc     r2, 5        // Write coprocessor register 5
```

Format 2

Definition:

```
   STC REG1, IM(255), OP(255)
```

Operation:

```
   Coprocessor Register IM(255) = ((unsigned32) REG1 & 0x00FFFFFF) | (OP(255)<<24)
```

Example:

```
   stc     r2, 5, 0xFF  // Write coprocessor register 5
                        // (force the MS byte to 0xFF)
   stc     r2, 5, r3.b0 // Write coprocessor register 5, with
                        // bits 31:24 coming from r3.b0
```

Clear Register Space (ZERO)

This pseudo-op is used to clear space in the register file (set to zero).

Definition:

```
    ZERO IM(123), IM(124)
    ZERO &REG1, IM(124)
```

Operation:

```
   The register file data starting at offset IM(123) (or &REG1) with a length of IM(124) is cleared to zero.
```

Example:

```
    zero    0, 8     // Set R0 and R1 to zero
    zero    &r0, 8   // Set R0 and R1 to zero
    // Set all elements in myStruct zero
    zero    &myStruct, SIZE(myStruct)
```

This pseudo-op will generate the necessary LDI instructions to clear the specified register range to zero. The instructions generated are optimized based on the starting register alignment and length.

Register Transfer In, Out, and Exchange (XIN, XOUT, XCHG)

These XFR pseudo-ops use the XFR wide transfer bus to read in a range of bytes into the register file, write out a range of bytes from the register file, or exchange the range of bytes to/from the register file.

Definition:

```
    XIN IM(253), REG, IM(124)
    XIN IM(253), REG, bn
    XOUT IM(253), REG, IM(124)
    XOUT IM(253), REG, bn
    XCHG IM(253), REG, IM(124)
    XCHG IM(253), REG, bn
```

Operation:

```
    On XIN, the register file data starting at the register REG with a
length of IM(124) is read in from the parallel XFR interface from the
hardware device with the device id specified in IM(253).
```

```
    On XOUT, the register file data starting at the register REG with a
 length of IM(124) is written out to the parallel XFR interface to the
hardware device with the device id specified in IM(253).
    On XCHG, the register file data starting at the register REG with a
 length of IM(124) is exchanged on the parallel XFR interface between
the register file and the hardware device with the device id specified
in IM(253).
```

Example:

```
    XIN XID_SCRATCH, R2, 8 // Read 8 bytes from scratch to R2:R3
    XOUT XID_SCRATCH, R2, b2 // Write 'b2' byte to scratch starting at R2
    XCHG XID_SCRATCH, R2, 8 // Exchange the values of R2:R3 with 8 bytes
    // from scratch
    XIN XID_PKTFIFO, R6, 24 // Read 24 bytes from the "Packet FIFO"
    // info R6:R7:R8:R9
```

Transfer Bus Hardware Connection

The transfer bus coming out of the PRU consists of 124 bytes of data and a sufficient number of control lines to control the transfer. Any given transfer will consist of a direction (in or out of the PRU), a peripheral ID, a starting byte offset, and a length. These can be represented in hardware as register and byte enable signals as needed for a proper implementation (which is beyond the scope of this description).

How the bus transfer is used is entirely up to the peripherals that connect to it. The number of registers that are implemented on the peripheral and how they align to the PRU register file is determined by the peripheral connection. For example, the system below connects PRU registers R1::R3 to "peripheral A" registers A0::A2, and connects PRU registers R2::R4 to "peripheral B" registers B0::B2.

Flow Control

Unconditional Jump (JMP)

Unconditional jump to a 16 bit instruction address, specified by register or immediate value.

Definition:

```
   JMP OP(65535)
```

Operation:

```
   PRU Instruction Pointer = OP(65535)
```

Example:

```
    jmp     r2.w0    // Jump to the address stored in r2.w0
    jmp     myLabel  // Jump to the supplied code label
```

Unconditional Jump and Link (JAL)

Unconditional jump to a 16 bit instruction address, specified by register or immediate value. The address following the JAL instruction is stored into REG1, so that REG1 can later be used as a "return" address.

Definition:

```
JAL REG1, OP(65535)
```

Operation:

```
REG1 = Current PRU Instruction Pointer + 1
PRU Instruction Pointer = OP(65535)
```

Example:

```
 jal     r2.w2, r2.w0      // Jump to the address stored in r2.w0
                           // put return location in r2.w2
 jal     r30.w0, myLabel   // Jump to the supplied code label and
                           // put the return location in r30.w0
```

Call Procedure (CALL)

The CALL instruction is a pseudo op designed to emulate a subroutine call on a stack based processor. Here, the JAL instruction is used with a specific call/ret register being the location to save the return pointer. The default register is R30.w0, but this can be changed by using the .setcallreg dot command. This instruction works in conjunction with the ".ret" dot command (deprecated) or the RET pseudo op instruction.

This instruction is not supported in the clpru assembler.

Definition:

```
CALL OP(65535)
```

Operation:

```
JAL call register, OP(65535)   (where call register defaults to r30.w0)
```

Example:

```
 call    r2.w0    // Call to the address stored in r2.w0
 call    myLabel  // Call to the supplied code label
```

Return from Procedure (RET)

The RET instruction is a pseudo op designed to emulate a subroutine return on a stack based processor. Here, the JMP instruction is used with a specific call/ret register being the location of the return pointer. The default register is R30.w0, but this can be changed by using the .setcallreg dot command. This instruction works in conjunction with the CALL pseudo op instruction.

This instruction is not supported in the clpru assembler.

Definition:

```
RET
```

Operation:

```
JMP call register   (where call register defaults to r30.w0)
```

Example:

```
    ret    // Return address stored in our call register
```

Quick Branch if Greater Than (QBGT)

Jumps if the value of OP(255) is greater than REG1.

Definition:

```
   QBGT LABEL, REG1, OP(255)
```

Operation:

```
   Branch to LABEL if OP(255) > REG1
```

Example:

```
    qbgt    myLabel, r2.w0, 5  // Branch if 5 > r2.w0
    qbgt    myLabel, r3, r4    // Branch if r4 > r3
```

Quick Branch if Greater Than or Equal (QBGE)

Jumps if the value of OP(255) is greater than or equal to REG1.

Definition:

```
   QBGE LABEL, REG1, OP(255)
```

Operation:

```
   Branch to LABEL if OP(255) >= REG1
```

Example:

```
    qbge    myLabel, r2.w0, 5  // Branch if 5 >= r2.w0
    qbge    myLabel, r3, r4    // Branch if r4 >= r3
```

Quick Branch if Less Than (QBLT)

Jumps if the value of OP(255) is less than REG1.

Definition:

```
   QBLT LABEL, REG1, OP(255)
```

Operation:

```
   Branch to LABEL if OP(255) < REG1
```

Example:

```
    qblt    myLabel, r2.w0, 5  // Branch if 5 < r2.w0
    qblt    myLabel, r3, r4    // Branch if r4 < r3
```

Quick Branch if Less Than or Equal (QBLE)

Jumps if the value of OP(255) is less than or equal to REG1.

Definition:

```
QBLE LABEL, REG1, OP(255)
```

Operation:

```
Branch to LABEL if OP(255) <= REG1
```

Example:

```
 qble    myLabel, r2.w0, 5  // Branch if 5 <= r2.w0
 qble    myLabel, r3, r4    // Branch if r4 <= r3
```

Quick Branch if Equal (QBEQ)

Jumps if the value of OP(255) is equal to REG1.

Definition:

```
QBEQ LABEL, REG1, OP(255)
```

Operation:

```
Branch to LABEL if OP(255) == REG1
```

Example:

```
 qbeq    myLabel, r2.w0, 5  // Branch if r2.w0==5
 qbeq    myLabel, r3, r4    // Branch if r4==r3
```

Quick Branch if Not Equal (QBNE)

Jumps if the value of OP(255) is NOT equal to REG1.

Definition:

```
QBNE LABEL, REG1, OP(255)
```

Operation:

```
Branch to LABEL if OP(255) != REG1
```

Example:

```
 qbne    myLabel, r2.w0, 5  // Branch if r2.w0!=5
 qbne    myLabel, r3, r4    // Branch if r4!=r3
```

Quick Branch Always (QBA)

Jump always. This is similar to the JMP instruction, only QBA uses an address offset and thus can be relocated in memory.

Definition:

```
QBA LABEL
```

Operation:

```
Branch to LABEL
```

Example:

```
 qba    myLabel        // Branch
```

Quick Branch if Bit is Set (QBBS)

Jumps if the bit OP(31) is set in REG1.

Format 1

Definition:

```
QBBS LABEL, REG1, OP(31)
```

Operation:

```
Branch to LABEL if( REG1 & ( 1 <<  (OP(31) & 0x1f) ) )
```

Example:

```
 qbbs     myLabel r3, r1     // Branch if( r3&(1<<r1) )
 qbbs     myLabel, r1.b1, 5  // Branch if( r1.b1 & 1<<5 )
```

Format 2

Definition:

```
QBBS LABEL, Rn.tx
```

Operation:

```
Branch to LABEL if( Rn & Rn.tx )
```

Example:

```
 qbbs     myLabel, r1.b1.t5  // Branch if( r1.b1 & 1<<5 )
 qbbs     myLabel, r0.t0     // Brach if bit 0 in R0 is set
```

Quick Branch if Bit is Clear (QBBC)

Jumps if the bit OP(31) is clear in REG1.

Format 1

Definition:

```
QBBC LABEL, REG1, OP(31)
```

Operation:

```
Branch to LABEL if( !(REG1 & ( 1 <<  (OP(31) & 0x1f) )) )
```

Example:

```
 qbbc     myLabel r3, r1     // Branch if( !(r3&(1<<r1)) )
 qbbc     myLabel, r1.b1, 5  // Branch if( !(r1.b1 & 1<<5) )
```

Format 2

Definition:

```
QBBC LABEL, Rn.tx
```

Operation:

```
Branch to LABEL if( !(Rn & Rn.tx) )
```

Example:

```
 qbbc     myLabel, r1.b1.t5      // Branch if( !(r1.b1 & 1<<5) )
 qbbc     myLabel, r0.t0    // Brach if bit 0 in R0 is clear
```

Wait until Bit Set (WBS)

The WBS instruction is a pseudo op that uses the QBBC instruction. It is used to poll on a status bit, spinning until the bit is set. In this case, REG1 is almost certainly R31, else this instruction could lead to an infinite loop.

Format 1

Definition:

```
WBS REG1, OP(31)
```

Operation:

```
QBBC $, REG1, OP(31)
```

Example:

```
 wbs     r31, r1           // Spin here while ( !(r31&(1<<r1)) )
 wbs     r31.b1, 5        // Spin here while ( !(r31.b1 & 1<<5) )
```

Format 2

Definition:

```
WBS Rn.tx
```

Operation:

```
QBBC $, Rn.tx
```

Example:

```
wbs     r31.b1.t5     // Spin here while ( !(r31.b1 & 1<<5) )
wbs     r31.t0        // Spin here while bit 0 in R31 is clear
```

Wait until Bit Clear (WBC)

The WBC instruction is a pseudo op that uses the QBBS instruction. It is used to poll on a status bit, spinning until the bit is clear. In this case, REG1 is almost certainly R31, else this instruction could lead to an infinite loop.

Format 1

Definition:

```
WBC REG1, OP(31)
```

Operation:

```
QBBS $, REG1, OP(31)
```

Example:

```
wbc     r31, r1     // Spin here while ( r31&(1<<r1) )
wbc     r31.b1, 5        // Spin here while ( r31.b1 & 1<<5 )
```

Format 2

Definition:

```
WBC Rn.tx
```

Operation:

```
QBBS $, Rn.tx
```

Example:

```
 wbc     r31.b1.t5    // Spin here while ( r31.b1 & 1<<5 )
 wbc     r31.t0       // Spin here while bit 0 in R31 is set
```

Halt Operation (HALT)

The HALT instruction disables the PRU. This instruction is used to implement software breakpoints in a debugger. The PRU program counter remains at its current location (the location of the HALT). When the PRU is re-enabled, the instruction is re-fetched from instruction memory.

Definition:

```
HALT
```

Operation:

```
Disable PRU
```

Example:

```
 halt
```

Sleep Operation (SLP)

The SLP instruction will sleep the PRU, causing it to disable its clock. This instruction can specify either a permanent sleep (requiring a PRU reset to recover) or a "wake on event". When the wake on event option is set to "1", the PRU will wake on any event that is enabled in the PRU Wakeup Enable register.

Definition:

```
SLP IM(1)
```

Operation:

```
Sleep the PRU with optional "wake on event" flag.
```

Example:

```
 SLP  0    // Sleep without wake events
 SLP  1    // Sleep until wake event set
```

Hardware Loop Assist (LOOP)

Defines a hardware-assisted loop operation. The loop is non-interruptible (LOOP). The loop operation works by detecting when the instruction pointer would normal hit the instruction at the designated target label, and instead decrementing a loop counter and jumping back to the instruction immediately following the loop instruction.

Definition:

```
LOOP LABEL, OP(256)
```

Operation:

```
LoopCounter = OP(256)
LoopTop = $+1
While (LoopCounter>0)
{
    If (InstructionPointer==LABEL)
    {
        LoopCounter--;
        InstructionPointer = LoopTop;
    }
}
```

Example 1:

```
    loop EndLoop, 5 // Peform the loop 5 times
    mvi r2, *r1.b0 // Get value
    xor r2, r2, r3 // Change value
    mvi *r1.b0++, r1 // Save value
    EndLoop:
```

Example 2:

```
    mvi r2, *r1.b0++ // Get the number of elements
    loop EndLoop, r2 // Peform the loop for each element
    mvi r2, *r1.b0 // Get value
    call ProcessValue // It is legal to jump outside the loop
    mvi *r1.b0++, r1 // Save value
    EndLoop:
```

Note: When the loop count is set from a register, only the 16 LS bits are used (regardless of the field size). If this 16-bit value is zero, the instruction jumps directly to the end of loop.

Return to Main Page on PRU Software Development

Click here.

References

[1] http://www.ti.com/lit/pdf/spruhv6

Article Sources and Contributors

PRU Assembly Instructions *Source*: http://processors.wiki.ti.com/index.php?oldid=226617 *Contributors*: A0272269, Caddison, D-allred, FarMcKon, Ipburbank, Jasonreeder, Kd5snu, M-watkins, Mtzgustavo, Tmauer