# Programmable Real-time Unit Subsystem Training Material

Aug 26, 2009

# Introduction

1. What is PRU SS?
   - **P**rogrammable **R**eal-time **U**nit **S**ub**S**ystem
   - Dual 32bit RISC processors running at ½ CPU freq.
   - Local instruction and data RAM. Access to SoC resources

2. What devices include PRU SS?
   - OMAPL138
   - C6748, C6746

3. Why PRU SS?
   - Full programmability allows adding customer differentiation
   - Efficient in performing embedded tasks that require manipulation of packed memory mapped data structures
   - Efficient in handling of system events that have tight realtime constraints.

# PRUSS Is/Is-Not

| Is | Is-Not |
|---|---|
| Dual 32-bit RISC processor specifically designed for manipulation of packed memory mapped data structures and implementing system features that have tight real time constraints | In not a H/W accelerator to speed up algorithm computations. |
| Simple RISC ISA<br>➢Approximately 40 instructions<br>➢Logical, arithmetic, and flow control ops all complete in a single cycle | Is not a general purpose RISC processor<br>➢No multiply hardware/instructions, no cache, no pipeline<br>➢No C programming |
| Simple tooling - basic command-line assembler/linker | Is not integrated with CCS. Doesn't include advanced debug options |
| Includes example code to demonstrate various features. Examples can be used as building blocks | No Operating System and high level application SW stack |

# PRU Value

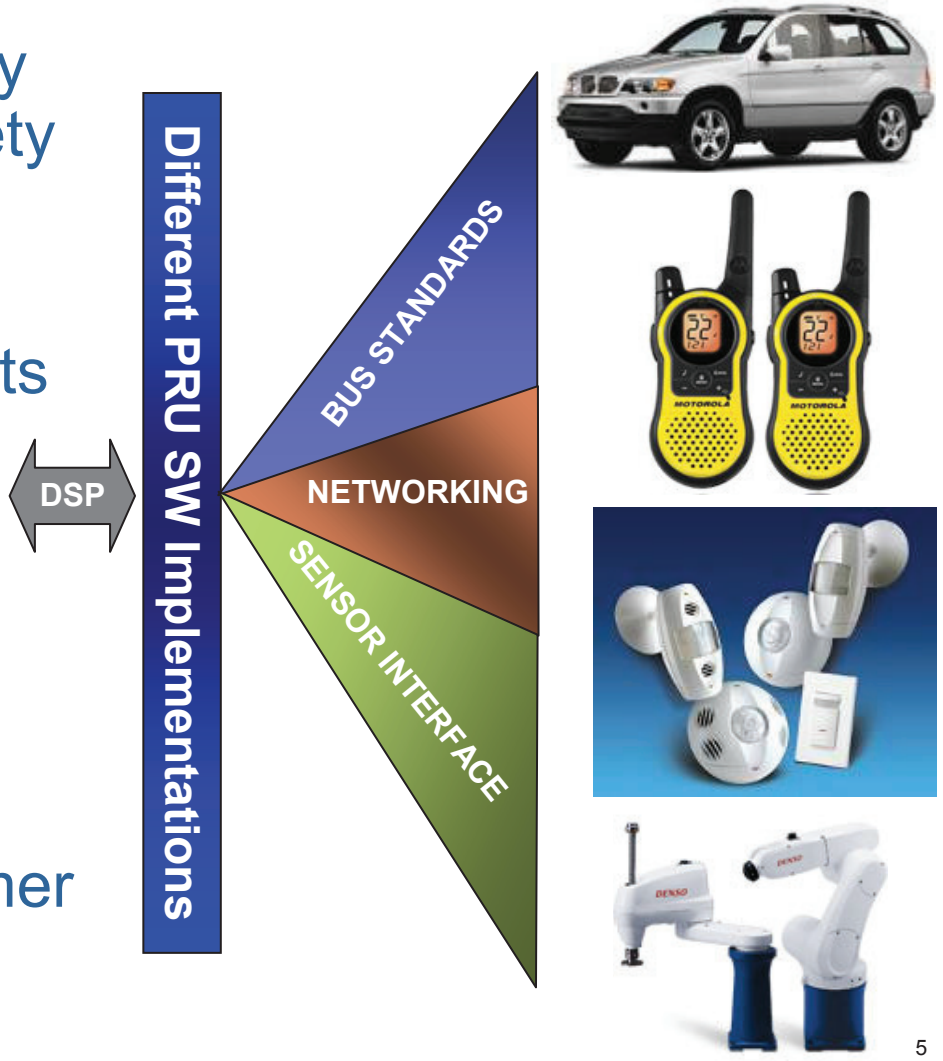
1. Extend Connectivity and Peripheral capability
   - Implement special peripherals and bus interfaces (e.g. UARTs)
   - Implement smart data movement schemes. Especially useful for Audio algorithms (e.g. Reverb, Room Correction)

2. Reduce System Power Consumption
   - Allows switching off both ARM and DSP clocks
   - Implement smart power controller by evaluating events before waking up DSP and/or ARM. Maximized power down time

3. Accelerate System Performance
   - Full programmability allows custom interface implementation
   - Specialized custom data handling to offload DSP for innovative signal processing algorithm implementation

# Special Interface Implementation

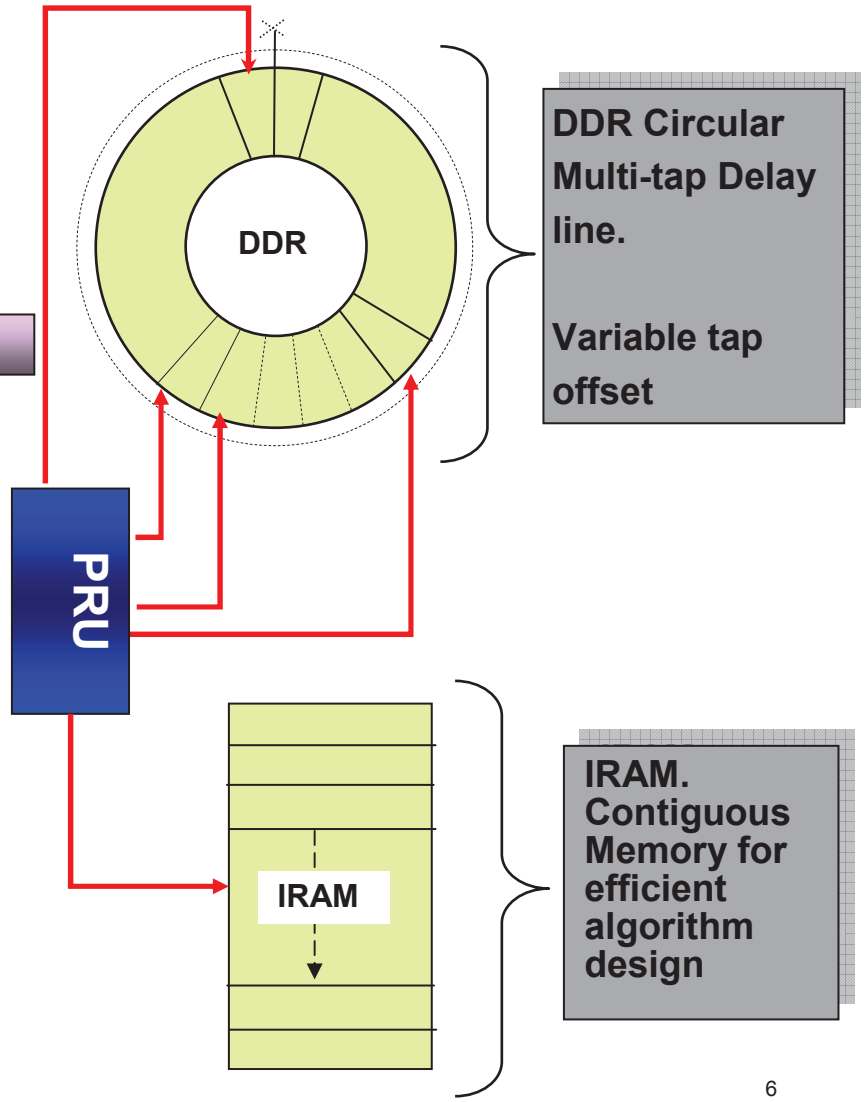
1. Extends SoC capability by allowing interface to variety of data sensors
2. Enables access to markets that require support for application specific interconnects and unique bus interfaces
3. Allows customer system differentiation and customer platform reuse



# Smart Data Move and Advanced DMA

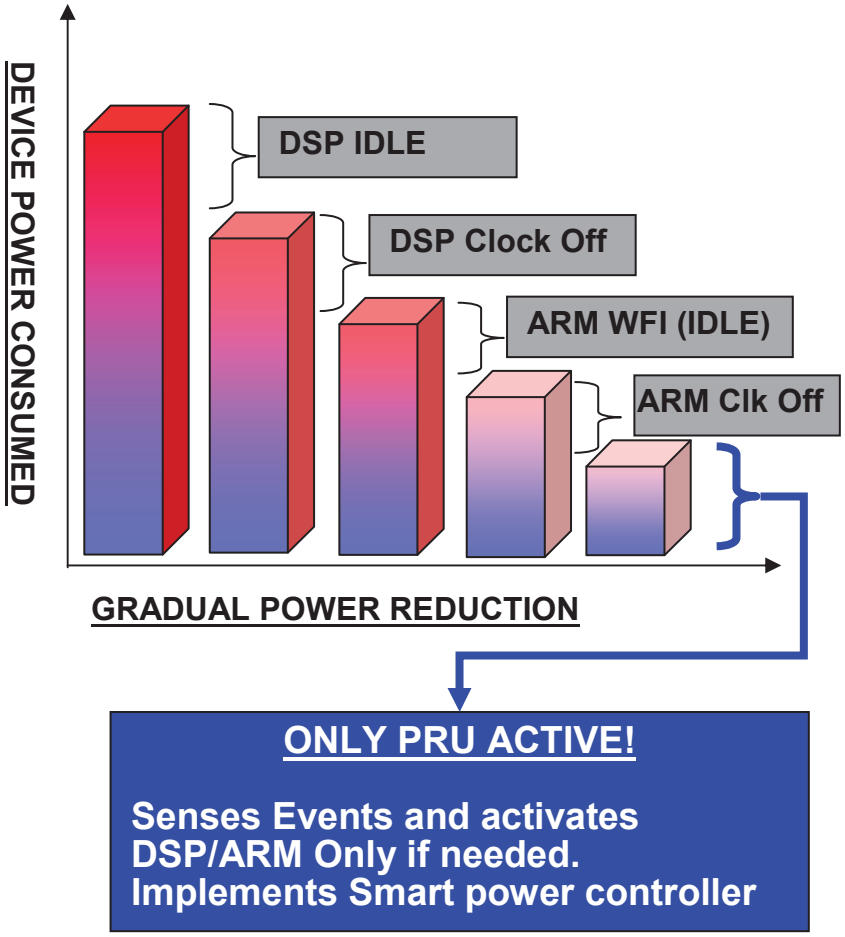
Musical Instruments; Pro-Audio Synthesizers; Audio Conferencing

1. Advanced DMA Operation
   - Simplifies audio algorithm development.
   - Low MIPS implementation of complex audio algorithm; Reverberation; Room Correction
   - On the fly data format modifications reduces CPU overhead
2. Smart Data Move
   - Buffer manipulation; data blending
   - Smart data rendering with various fill effects



# Extends Low Power Advantage

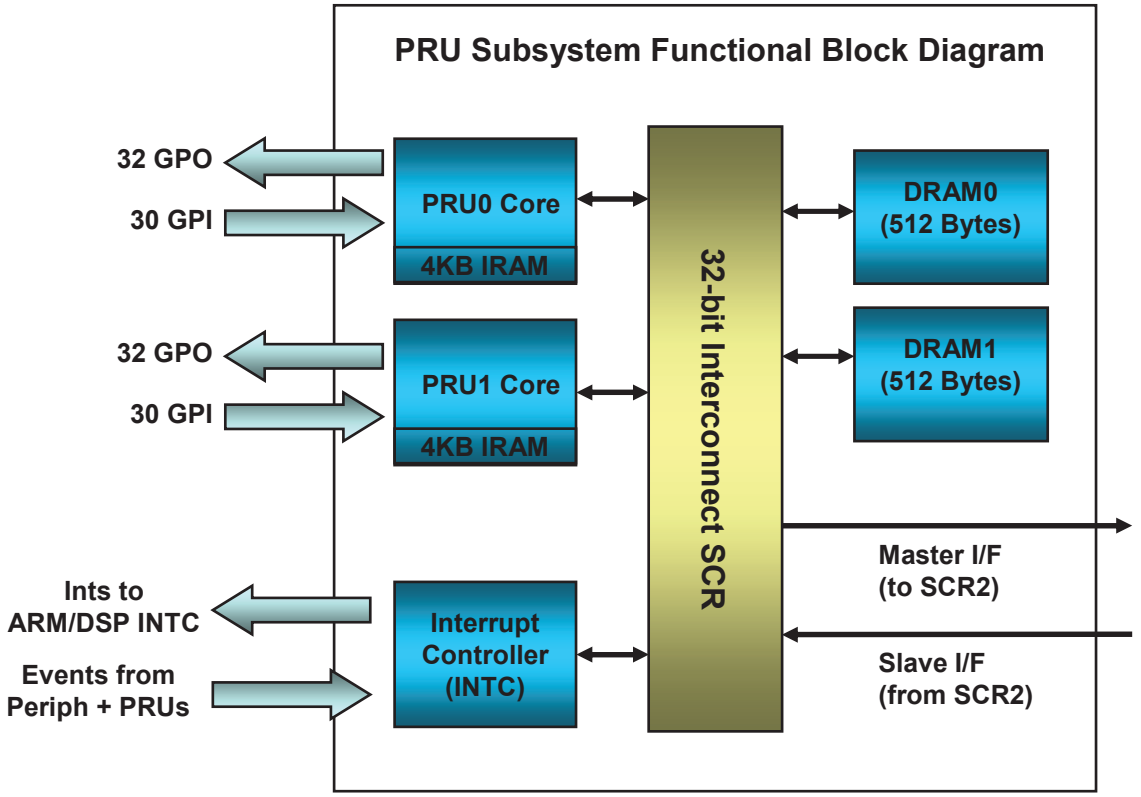
1. Capable to receive majority of system events (up to 32 at a time).
2. Can put both ARM and DSP to lowest power modes; ie Turn off clocks to both the processors
3. Implement smart power controller by analyzing events and only enabling DSP/ARM for relevant events. Maximizes the power down state
4. SW Programmable to handle tasks for common events; thereby reducing need to activate DSP/ARM



# PRU Subsystem Overview

# PRU Subsystem

- Provides two independent programmable real-time (PRU) cores
  - 32-Bit Load/Store RISC architecture
  - 4K Byte instruction RAM (1K instructions) per core
  - 512 Bytes data RAM per core
- PRU operation is little endian similar to ARM and DSP processors
- Includes Interrupt Controller for system event handling
- Fast I/O interface
  - 30 input pins and 32 output pins per PRU core
- Power management via single PSC



# Local & Global Memory Map

- Local Memory Map
  - Allows PRU to directly access subsystem resources, e.g. DRAM, INTC registers, etc.
  - NOTE: Memory map slightly different from PRU0 and PRU1 point-of-view.
- Global Memory Map
  - Allows external masters to access PRU subsystem resources, e.g. debug and control registers.
  - PRU cores can also use global memory map, but more latency since access routed through SCR2.

Instruction Space Memory Map

| Start | End | PRU0 | PRU1 |
|---|---|---|---|
| 0x00000000 | 0x00000FFF | PRU0 Instruction RAM | PRU1 Instruction RAM |

Data Space Memory Map

| Start | End | PRU0 | PRU1 |
|---|---|---|---|
| 0x00000000 | 0x000001FF | Data RAM 0 | Data RAM 1 |
| 0x00000200 | 0x00001FFF | Reserved | Reserved |
| 0x00002000 | 0x000021FF | Data RAM 1 | Data RAM 0 |
| 0x00002200 | 0x00003FFF | Reserved | Reserved |
| 0x00004000 | 0x00006FFF | INTC Registers | INTC Registers |
| 0x00007000 | 0x000073FF | PRU0 Control Registers | PRU0 Control Registers |
| 0x00007400 | 0x000077FF | Reserved | Reserved |
| 0x00007800 | 0x00007BFF | PRU1 Control Registers | PRU1 Control Registers |
| 0x00007C00 | 0x0000FFFF | Reserved | Reserved |
| 0x00010000 | 0xFFFFFFFF | Reserved | Reserved |

Global Address Map

| Registers | | |
|---|---|---|
| Address Offset | | Region |
| 0x01C30000 | 0x01C301FF | Data RAM 0 |
| 0x01C30200 | 0x01C31FFF | Reserved |
| 0x01C32000 | 0x01C321FF | Data RAM 1 |
| 0x01C32200 | 0x01C33FFF | Reserved |
| 0x01C34000 | 0x01C36FFF | INTC Registers |
| 0x01C37000 | 0x01C373FF | PRU0 Control Registers |
| 0x01C37400 | 0x01C377FF | PRU0 Debug Registers |
| 0x01C37800 | 0x01C37BFF | PRU1 Control Registers |
| 0x01C37C00 | 0x01C37FFF | PRU1 Debug Registers |
| 0x01C38000 | 0x01C38FFF | PRU0 Instruction RAM |
| 0x01C39000 | 0x01C3BFFF | Reserved |
| 0x01C3C000 | 0x01C3CFFF | PRU1 Instruction RAM |
| 0x01C3D000 | 0x01C3FFFF | Reserved |

# PRU Overview

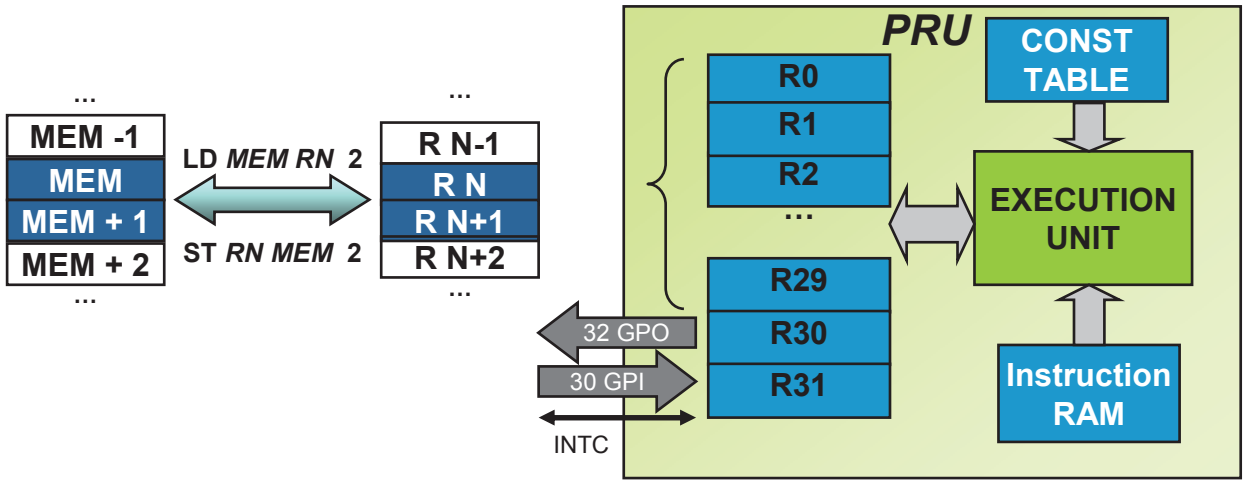
# PRU Functional Block Diagram

## General Purpose Registers

- All instructions are performed on registers and complete in a single cycle
- Register file appears as linear block for all register to memory operations

## Constant Table

- Ease SW development by providing freq used constants
- Peripheral base addresses
- Few entries programmable




## Execution Unit

- Logical, arithmetic, and flow control instructions
- Scalar, no Pipeline, Little Endian
- Register-to-register data flow
- Addressing modes: Ld Immediate & Ld/St to Mem

## Special Registers (R30 and R31)

- R30
  - Write: 32 GPO
- R31
  - Read: 30 GPI + 2 Host Int status
  - Write: Generate INTC Event

## Instruction RAM

- 4KB in size; 1K Instructions
- Can be updated with PRU reset

# PRU Constants Table

- Load and store instructions require that the destination/source base address be loaded in a register.
- Constants table is a list of 32 commonly-used addresses that can be used in memory load and store operations via special instructions.
- Most constant table entries are fixed, but some contain a programmable bit field that is programmable through the PRU control registers.
- Using the constants table saves both the register space as well as the time required to load pointers into registers.

# PRU0/1 Constants Table

| Entry # | Region Pointed To | Value [31:0] |
|---|---|---|
| 0 | PRU INTC | 0x00004000 |
| 1 | Timer64P0 | 0x01C20000 |
| 2 | I2C0 | 0x01C22000 |
| 3 | PRU0/1 Local Data | 0x00000000 |
| 4 | PRU1/0 Local Data | 0x00002000 |
| 5 | MMC/SD | 0x01C40000 |
| 6 | SPI0 | 0x01C41000 |
| 7 | UART0 | 0x01C42000 |
| 8 | McASP0 DMA | 0x01D02000 |
| 9 | *RESERVED* | *0x01D06000* |
| 10 | *RESERVED* | *0x01D0A000* |
| 11 | UART1 | 0x01D0C000 |
| 12 | UART2 | 0x01D0D000 |
| 13 | USB0 | 0x01E00000 |
| 14 | USB1 | 0x01E25000 |
| 15 | UHPI Config | 0x01E10000 |

| Entry # | Region Pointed To | Value [31:0] |
|---|---|---|
| 16 | *RESERVED* | *0x01E12000* |
| 17 | I2C1 | 0x01E28000 |
| 18 | EPWM0 | 0x01F00000 |
| 19 | EPWM1 | 0x01F02000 |
| 20 | *RESERVED* | *0x01F04000* |
| 21 | ECAP0 | 0x01F06000 |
| 22 | ECAP1 | 0x01F07000 |
| 23 | ECAP2 | 0x01F08000 |
| 24 | PRU0/1 Local Data | 0x00000n00, n = c24_blk_index[3:0] |
| 25 | McASP0 Control | 0x01D00n00, n = c25_blk_index[3:0] |
| 26 | *RESERVED* | *0x01D04000* |
| 27 | *RESERVED* | *0x01D08000* |
| 28 | DSP RAM/ROM | 0x11nnnn00, nnnn = c28_pointer[15:0] |
| 29 | EMIFa SDRAM | 0x40nnnn00, nnnn = c29_pointer[15:0] |
| 30 | L3 RAM | 0x80nnnn00, nnnn = c30_pointer[15:0] |
| 31 | EMIFb Data | 0xC0nnnn00, nnnn = c31_pointer[15:0] |

**NOTES**

**1. Constants not in this table can be created 'on the fly' by loading two 16-bit values into a PRU register. These constants are just ones that are expected to be commonly used, enough so to be hard-coded in the PRU constants table.**

**2. Constants table entries 24 through 31 are not fully hard coded, they contain a programmable bit field that is programmable through the PRU control registers. Programmable entries allow you to select different 256-byte pages within an address range.**

# PRU Event/Status Register (R31)

- Writes: Generate output events to the INTC.
  - Write the event number (0 through 31) to PRU_VEC[4:0] and simultaneously set PRU_VEC_VALID to create a pulse to INTC.
  - Outputs from both PRUs are ORed together to form single output.
  - Output events 0 through 31 are connected to system events 32 through 63 on INTC.
- Reads: Return Host 1 & 0 interrupt status from INTC and general purpose input pin status.

**R31 During Writes**

| Bit | Name | Description |
|---|---|---|
| 31:6 | RSV | Reserved |
| 5 | PRU_VEC_VALID | Valid strobe for vector output |
| 4:0 | PRU_VEC[4:0] | Vector output |

**R31 During Reads**

| Bit | Name | Description |
|---|---|---|
| 31 | PRU_INTR_IN[1] | PRU Host 1 interrupt from INTC |
| 30 | PRU_INTR_IN[0] | PRU Host 0 interrupt from INTC |
| 29:0 | PRU_R31_STATUS[29:0] | Status inputs from PRU*n*_R31[29:0] |

# Dedicated GPIs and GPOs

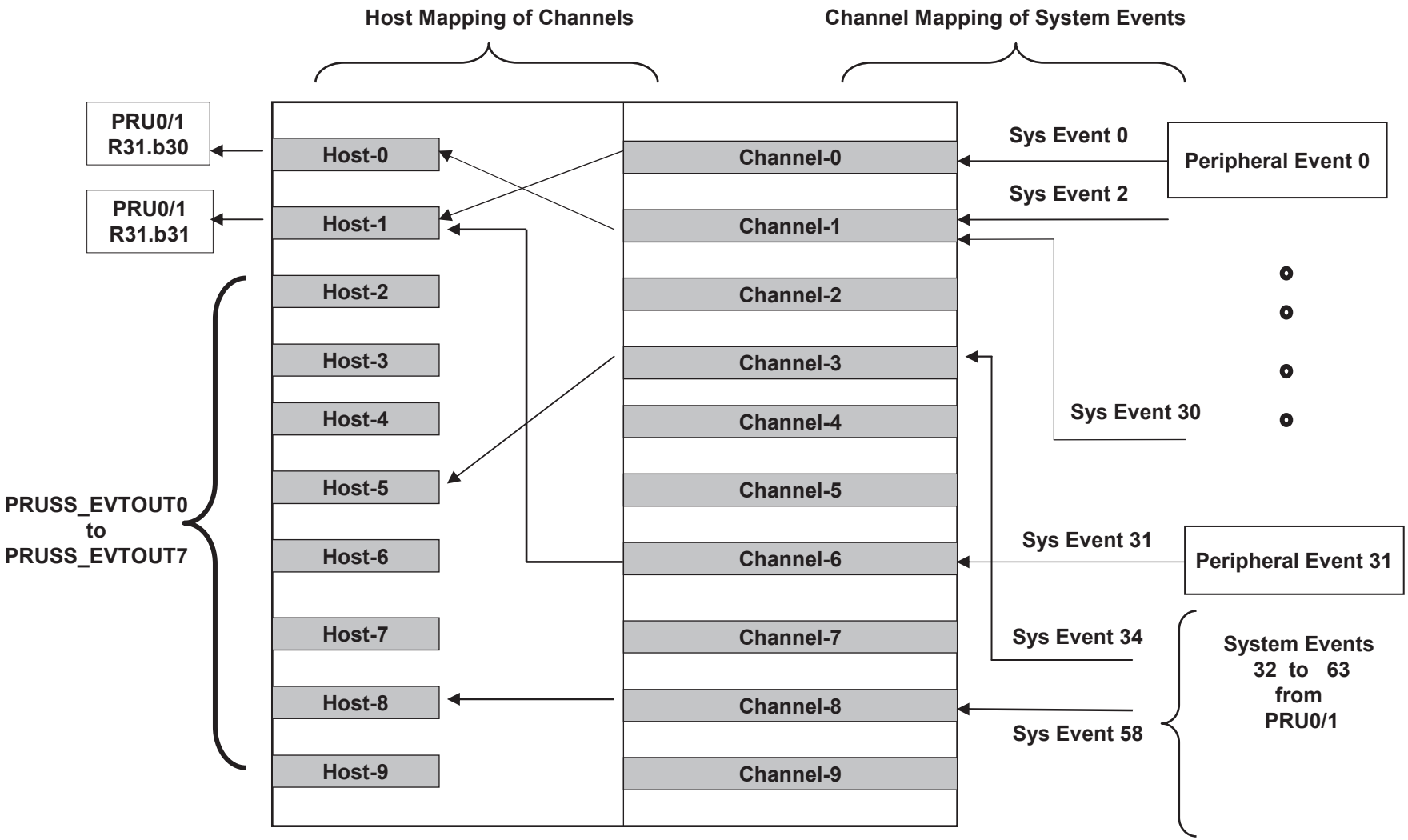
- General purpose inputs (GPIs)
  - Each PRU has ***30*** general purpose input pins: PRU0_R31[29:0] and PRU1_R31[29:0].
  - Reading R31[29:0] in each PRU returns the status of PRU*n*_R31[29:0].
- General purpose outputs (GPOs)
  - Each PRU has ***32*** general purpose output pins: PRU0_R30[31:0] and PRU1_R30[31:0].
  - The value written to R30[31:0] is driven on PRU*n*_R30[31:0].
- Notes
  - Unlike the device GPIOs, PRU GPIs and GPOs are assigned to different pins.
  - You can use the "." operator to read or write a single bit in R30 and R31, e.g. R30.t0.
  - PRU GPOs and GPIs are enabled through the system pin mux registers (PINMUX0-19).

# Interrupt Controller

# Interrupt Controller (INTC) Overview

- Supports 64 system events
  - 32 system events external to the PRU subsystem
  - 32 system events generated directly by the PRU cores
- Supports up to 10 interrupt channels
  - Allows for interrupt nesting.
- Generation of 10 host interrupts
  - Host Interrupt 0 mapped to R31.b30 in both PRUs
  - Host Interrupt 1 mapped to R31.b31 in both PRUs
  - Host Interrupt 2 to 9 routed to ARM and DSP INTCs.
- System events can be individually enabled, disabled, and manually triggered
- Each host event can be enabled and disabled
- Hardware prioritization of system events and channels

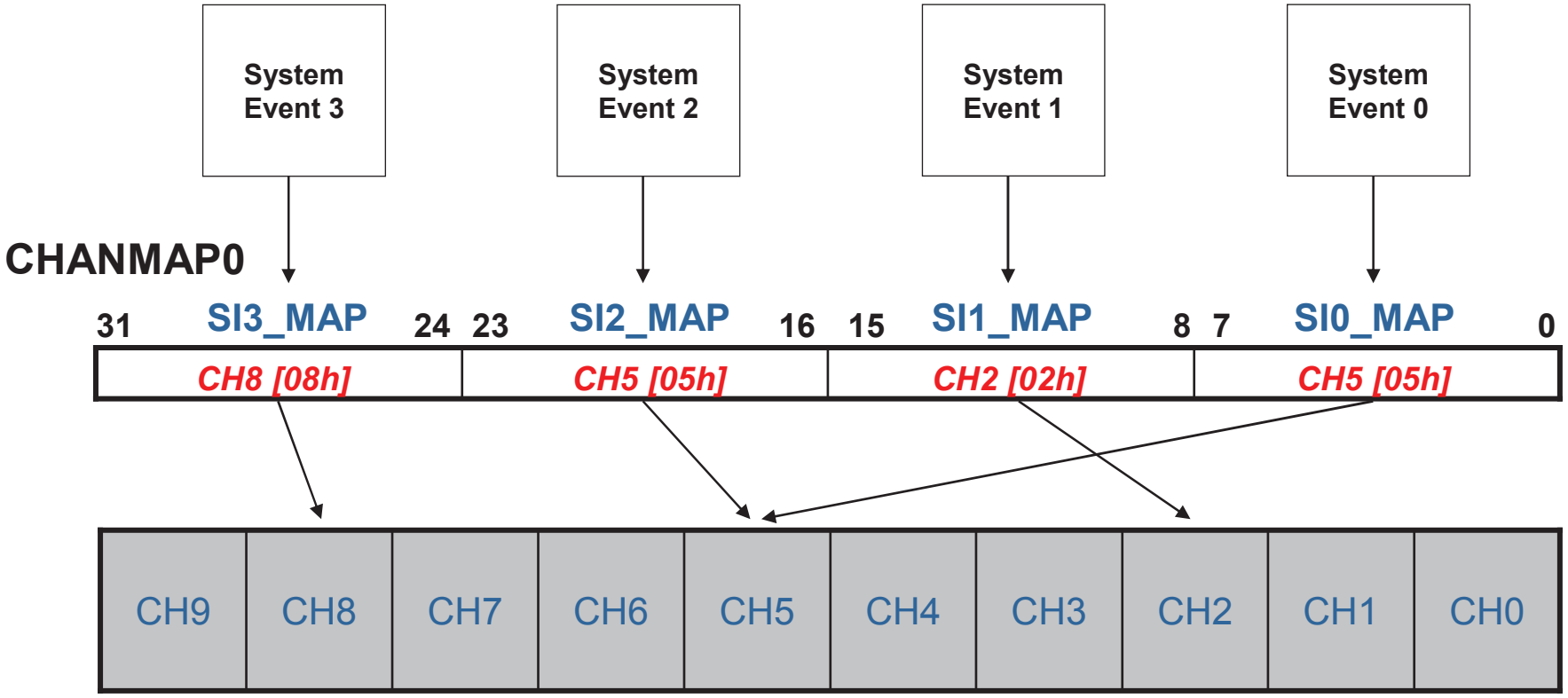
# Interrupt Controller Block Diagram



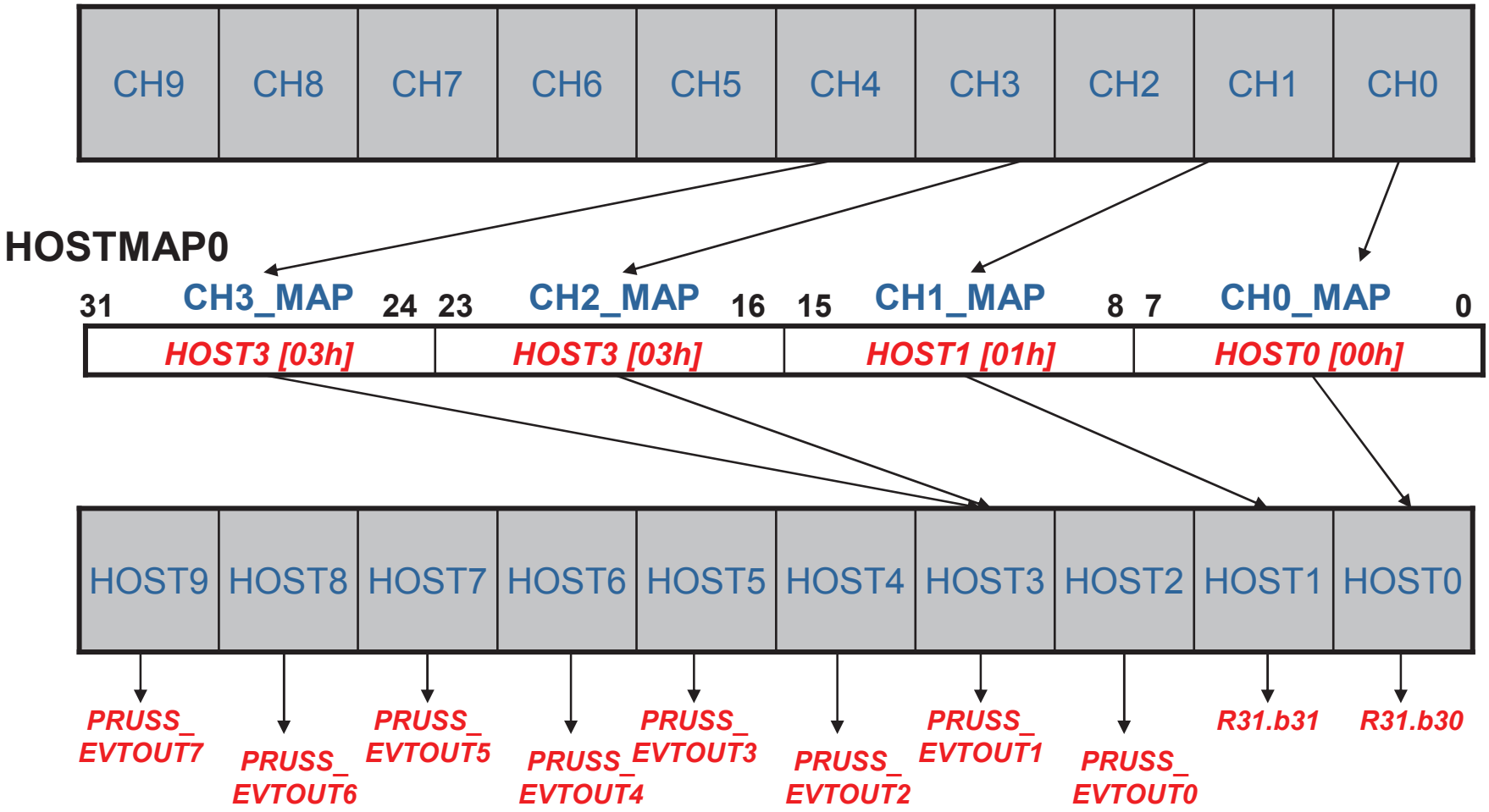
# Interrupt Controller Mapping

- System events must be mapped to channels
  - Multiple system events can be mapped to the same channel.
  - Not possible to map system events to more than one channel.
  - System events mapped to same channel ➔ lower-numbered events have higher priority
- Channels must be mapped to host interrupts
  - Multiple channels can be mapped to the same host interrupt.
  - Not possible to map channels to more than one host interrupt.
  - Recommended to map channel “x” to host interrupt “x”, where “x” is from 0 to 9.
  - Channels mapped to the same host interrupt ➔ lower-numbered channels have higher priority

# System Event to Channel Mapping



# Channel to Host Interrupt Mapping




*** Recommended to map channel "x" to host interrupt "x".**

# PRU Instruction Set

# PRU Instruction Overview

- Four instruction classes
  - Arithmetic
  - Logical
  - Flow Control
  - Register Load/Store
- Instruction Syntax
  - Mnemonic, followed by comma separated parameter list
  - Parameters can be a register, label, immediate value, or constant table entry
  - Example
    - SUB r3, r3, 10
    - Subtracts immediate value 10 (decimal) from the value in r3 and then places the result in r3
- Nearly all instructions (with exception of memory accesses) are single-cycle execute
  - 6.67 ns when running at maximum 150 MHz

# PRU Instruction Syntax Conventions

- Instruction definitions use a certain syntax to indicate acceptable parameters types

| 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 | r0.t23, r1.b2.t5 |
| Cn, Cn1, Cn2, … | Any 32 bit constant table entry (c0 through c31) | c0,c1 |
| bn | Specifies a field that must be b0, b1, b2, or b3 – denoting r0.b0, r0.b1, r0.b2, and r0.b3 respectively. | b0,b1 |
| 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. Immediate values can be specified with or without a leading hash "\#".Immediate values, labels, and register addresses are all acceptable. | #23, 0b0110, 2+2, &r3.w2, |
| OP(n) | The union of REG and IM(n) | r0, r1.w0, #0x7F, 1<<3, loop1, &r1.w0 |

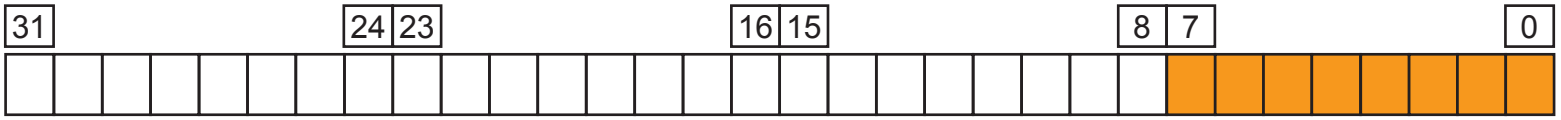
# PRU Register Accesses

- PRU is suited to handling packets and structures, parsing them into fields and other smaller data chunks
- Valid registers formats allow individual selection of bits, bytes, and half-words from within individual registers
- The parts of the register can be accessed using the modifier suffixes shown

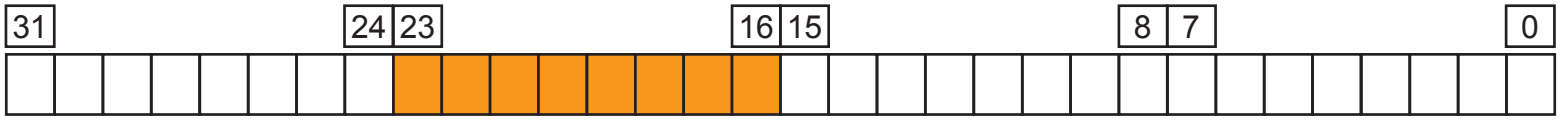
| Suffix | Range of n | Meaning |
|---|---|---|
| .wn | 0 to 2 | 16 bit field with a byte offset of *n* within the parent field |
| .bn | 0 to 3 | 8 bit field with a byte offset of *n* within the parent field |
| .tn | 0 to 31 | 1 bit field with a bit offset of *n* within the parent field |

# Register Examples

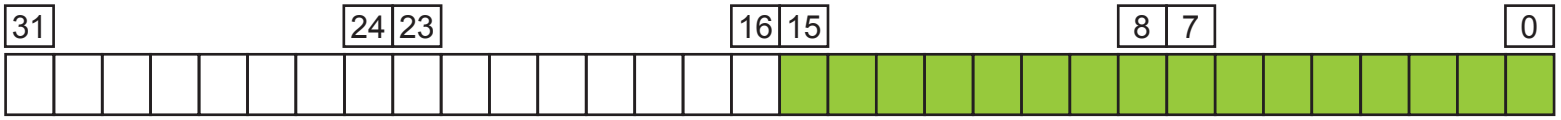
- r0.b0



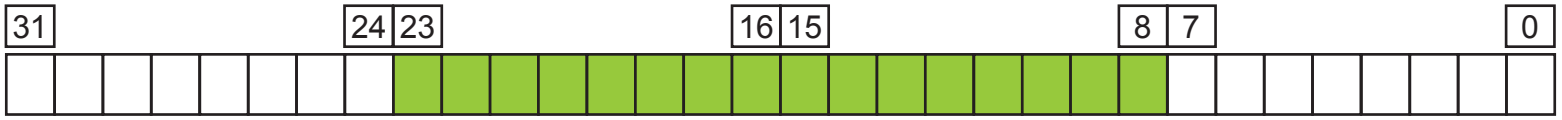

- r0.b2




- r0.w0

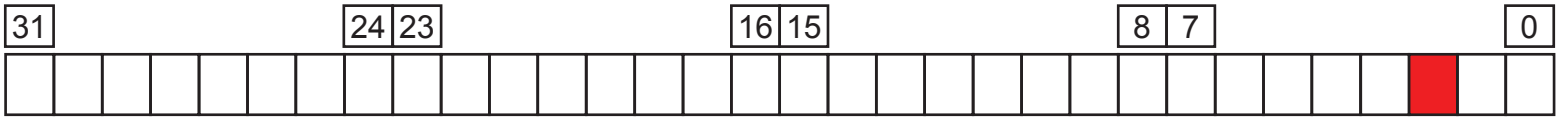



- r0.w1

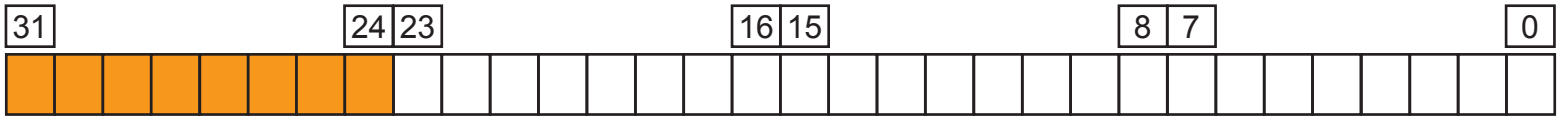


# Register Examples, cont'd

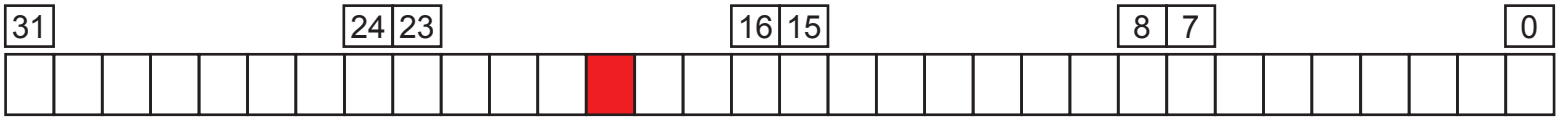
- r0.t2



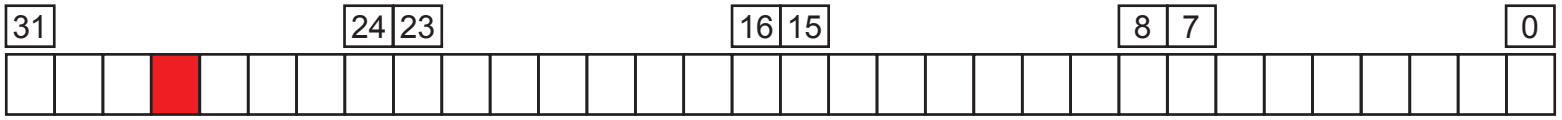

- r0.w2.b1 = r0.b3




- r0.w1.b1.t3 = r0.b2.t3 = r0.t19




- r0.w2.t12 = r0.t28



# PRU Instruction Set

**Arithmetic Operations (green)** **Logic Operations (blue)**

**IO Operations (black)** **Program Flow Control (red)**

***Pseudo* *Op-code (Italic)***

| | | | | |
|---|---|---|---|---|
| ADD | ADC | SUB | SUC | RSB |
| RSC | LSL | LSR | AND | OR |
| XOR | NOT | MIN | MAX | CLR |
| SET | SCAN | LMBD | *MOV* | LDI |
| LBBO | SBBO | LBCO | SBCO | *ZERO* |
| *MVIB* | *MVIW* | *MVID* | JAL | JMP |
| QBGT | QBGE | QBLT | QBLE | QBEQ |
| QBNE | QBA | QBBS | QBBC | *WBS* |
| *WBC* | HALT | SLP | *CALL* | *RET* |

# Arithmetic Instructions

- Unsigned Integer Add (ADD)
  - Performs 32-bit add on two 32 bit zero extended source values.
  - Definition:
    ADD REG1, REG2, OP(255)
  - Operation:
    REG1 = REG2 + OP(255)
    carry = (( REG2 + OP(255) ) >> *bitwidth*(REG1)) & 1
- 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
- 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
- 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

# Arithmetic Instructions, cont'd

- 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
- 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

# Logical Instructions

- 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)
- 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)
- 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)
- Bitwise NOT (NOT)
  - Performs 32-bit logical NOT on the 32 bit zero extended source value.
  - Definition:
    NOT REG1, REG2
  - Operation:
    REG1 = ~REG2

# Logical Instructions, cont'd

- 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 )
- 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 )
- 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);
- 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;

# Logical Instructions, cont'd

- 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)
  - Format 1 Operation:
    REG1 = REG2 & ~( 1 << (OP(31) & 0x1f) )
  - Format 2 (same source and destination) Definition:
    CLR REG1, OP(255)
  - Format 2 (same source and destination) Operation:
    REG1 = REG1 & ~( 1 << (OP(31) & 0x1f) )
  - Format 3 (source abbreviated) Definition:
    CLR REG1, Rn.tx
  - Format 3 (source abbreviated) Operation:
    REG1 = Rn & ~(1<<x)
  - Format 4 (same source and destination – abbreviated) Definition:
    CLR Rn.tx
  - Format 4 (same source and destination – abbreviated) Operation:
    Rn = Rn & ~(1<<x)
- Set Bit (SET)
  - Sets the specified bit in the source and copies the result to the destination. Various calling formats are supported.
  - Format 1 Definition:
    SET REG1, REG2, OP(31)
  - Format 1 Operation:
    REG1 = REG2 | ( 1 << (OP(31) & 0x1f) )
  - Format 2 (same source and destination) Definition:
    SET REG1, OP(31)
  - Format 2 (same source and destination) Operation:
    REG1 = REG1 | ( 1 << (OP(31) & 0x1f) )
  - Format 3 (source abbreviated) Definition:
    SET REG1, Rn.tx
  - Format 3 (source abbreviated) Operation:
    REG1 = Rn | (1<<x)
  - Format 4 (same source and destination – abbreviated) Definition:
    SET Rn.tx
  - Format 4 (same source and destination – abbreviated) Operation:
    Rn = Rn | (1<<x)

# Logical Instructions, cont'd

- 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;
```

# Flow Control Instructions

- 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)
- 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)
- 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
- 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.

# Flow Control Instructions, cont'd

- 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
- 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
- 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
- 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

# Flow Control Instructions, cont'd

- Quick Branch if Equal (QBEQ)
  - Jumps if the value of OP(255) is equal to REG1.
  - Definition:
    QBGT LABEL, REG1, OP(255)
  - Operation:
    Branch to LABEL if OP(255) == REG1
- 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
- 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

# Flow Control Instructions, cont'd

- 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)
  - Format 1 Operation:
    Branch to LABEL if( REG1 & ( 1 << (OP(31) & 0x1f) ) )
  - Format 2 Definition:
    QBBS LABEL, Rn.tx
  - Format 2 Operation:
    Branch to LABEL if( Rn & (1<<x) )
- 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)
  - Format 1 Operation:
    Branch to LABEL if( !(REG1 & ( 1 << (OP(31) & 0x1f) )) )
  - Format 2 Definition:
    QBBC LABEL, Rn.tx
  - Format 2 Operation:
    Branch to LABEL if( !(Rn & (1<<x)) )

# Load/Store Instructions

- 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)
- 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.
  - Format 1 (immediate count) definition:
    LBBO REG1, Rn2, OP(255), IM(124)
  - Format 1 (immediate count) operation:
    memcpy( offset(REG1), Rn2+OP(255), IM(124) );
  - Format 2 (register count) definition:
    LBBO REG1, Rn2, OP(255), bn
  - Format 2 (register count) operation:
    memcpy( offset(REG1), Rn2+OP(255), R0.bn );
- 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.
  - Format 1 (immediate count) definition:
    - SBBO REG1, Rn2, OP(255), IM(124)
  - Format 1 (immediate count) operation:
    - memcpy( Rn2+OP(255), offset(REG1), IM(124) );
  - Format 2 (register count) definition:
    - SBBO REG1, Rn2, OP(255), bn
  - Format 2 (register count) operation:
    - memcpy( Rn2+OP(255), offset(REG1), R0.bn );

# Load/Store Instructions, cont'd

- 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.
  - Format 1 (immediate count) definition:
    LBCO REG1, Cn2, OP(255), IM(124)
  - Format 1 (immediate count) operation:
    memcpy( offset(REG1), Cn2+OP(255), IM(124) );
  - Format 2 (register count) definition:
    LBCO REG1, Cn2, OP(255), bn
  - Format 2 (register count) operation:
    memcpy( offset(REG1), Cn2+OP(255), R0.bn );
- 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.
  - Format 1 (immediate count) definition:
    SBCO REG1, Cn2, OP(255), IM(124)
  - Format 1 (immediate count) operation:
    memcpy( Cn2+OP(255), offset(REG1), IM(124) );
  - Format 2 (register count) definition:
    SBCO REG1, Cn2, OP(255), bn
  - Format 2 (register count) operation:
    memcpy( Cn2+OP(255), offset(REG1), R0.bn );

# PASM assembler tool

# PASM Overview

- PASM is a command-line assembler for the PRU cores
  - Converts PRU assembly source files to loadable binary data
  - Output format can be raw binary, C array (default), or hex
  - Other debug formats also can be output
- Command line syntax:

  pasm [-bcmldxz] SourceFile [-Dname=value] [-CArrayname]
- The PASM tool generates a single monolithic binary
  - No linking, no sections, no memory maps, etc.
  - Code image begins at start of IRAM (offset 0x0000)

# Valid Assembly File Inputs

- Four basic assembler statements
  - Hash commands
  - Dot commands (directives)
  - Labels
  - Instructions
    - True instructions (defined previously)
    - Pseudo-instructions
- Assembly comments allowed and ignored
  - Use the double slash single-line format of C/C++
  - Always appear as last field on a line
  - Example:

```
//-------------------------
// This is a comment
//-------------------------
ldi r0, 100 // This is a comment
```

# Assembler Hash statements

- Similar to C pre-processor commands
- #include”filename”
  - Specified filename is immediately opened, parsed, and processed
  - Allows splitting large PRU assembly code into separate files
- #define
  - Specify a simple text substitution
  - Can also be used to define empty substitution for use with #ifdef, #ifndef, etc.
- #undef – Used to undefine a substitution previously defined with #define
- Others (#ifdef, #ifndef, #else, #endif, #error) as used in C preprocessor

# Assembler Dot Commands

- All dot commands start with a period (the dot)
- Rules for use
  - Must be only assembly statement on line
  - Can be followed by comments
  - Not required to start in column 0

| Command | Description |
|---|---|
| .origin | Set start of next assembly statement |
| .entrypoint | Only used for debugger, specifies starting address |
| .setcallreg | Specified 16-bit register field for storing return pointer |
| .macro, .mparam, .endm | Define assembler macros |
| .struct, .ends, .u32, .u16, .u8 | Define structure types for easier register allocation |
| .assign | Map defined structure into PRU register file |
| .enter | Create and enter new variable scope |
| .leave | Leave a specific variable scope |
| .using | Use a previously created and left scope |

# Macro Example

- PASM macros using dot commands expand are like C preprocessor macros using #define
- They save typing and can make code cleaner
- Common macro:

```
//
// mov32 : Move a 32bit value to a register
//
// Usage:
// mov32 dst, src
//
// Sets dst = src. Src must be a 32 bit immediate value.
//
.macro MOV32
.mparam dst, src
  LDI dst.w0, src & 0xFFFF
  LDI dst.w2, src >> 16
.endm
```

- Macro invoked as:

```
MOV32 r0, 0x12345678
```

# Struct Example

- Like in C, defined structures can be useful for defining offsets and mapping data into registers/memory
- Declared similar to using typedef in C
  - PASM automatically processes each declared structure template and creates an internal structure type.
  - The named structure type is not yet associated with any registers or storage.

- Example from C:

```
typedef struct _PktDesc_
{
  struct _PktDesc *pNext;
  char *pBuffer;
  unsigned short Offset;
  unsigned short BufLength;
  unsigned short Flags;
  unsigned short PktLength;
} PKTDESC;
```

- Now in PASM assembly:

```
.struct PktDesc
  .u32 pNext
  .u32 pBuffer
  .u16 Offset
  .u16 BufLength
  .u16 Flags
  .u16 PktLength
.ends
```

# Struct Example, cont'd

- To use the created structure type, we use .assign statement to map a region of the register file for use with struct syntax
- Example, using previously defined struct:
  .assign PktDesc, R4, R7, RxDesc
- When PASM sees this assignment, it will perform three tasks:
  - Verify that the structure perfectly spans the declared range (in this case R4 through R7). The application developer can avoid the formal range declaration by substituting '*' for 'R7' above.
  - Verify that all structure fields are able to be mapped onto the declared range without any alignment issues. If an alignment issue is found, it is reported as an error along with the field in question. Note that assignments can begin on any register boundary.
  - Create an internal data type named "RxDesc", which is of type "PktDesc".
- For the above assignment, variable to register mapping is as shown
- Using .struct and .assign means only a single code change if we want to relocate the variables in the register file

| Variable | Register Assignment |
|---|---|
| RxDesc | R4 |
| RxDesc.pNext | R4 |
| RxDesc.pBuffer | R5 |
| RxDesc.Offset | R6.w0 |
| RxDesc.BufLength | R6.w2 |
| RxDesc.Flags | R7.w0 |
| RxDesc.PktLength | R7.w2 |

# Labels

- Labels are used denote program addresses. When placed at the beginning of a source line and immediately followed by a colon ':', they mark a program address location
- When referenced by an instruction, the corresponding marked address is substituted for the label
- The rules for labels are as follows:
  - A label *definition* must be immediately followed by a colon
  - Only instructions and/or comments can occupy the same source line as a label
  - Labels can use characters A-Z, a-z, 0-9,underscores,and periods
  - A label can not begin with a number (0-9)
- Example:

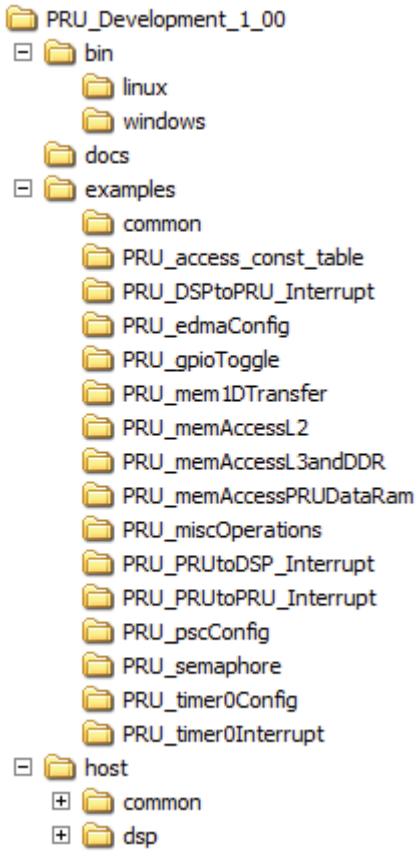
```
        LDI r0, 100
loop_label:
        SUB r0, r0, 1
        QBNE loop_label, r0, 0
        RET
```

# Instructions vs Psuedo-instructions

- Directly supported hardware instructions detailed previously
- PASM also supports a number of psuedo-instructions that expand to true hardware instructions
  - Copy Value (MOV)
  - Clear Register Space (ZERO)
  - Wait until Bit Set (WBS)
  - Wait until Bit Clear (WBC)
  - Call Procedure (CALL)
  - Return from Procedure (RET)

# Brief Overview of Software Package

# Package Contents




- Bin directory
  - PASM binary tool
- Doc
  - Current documentation
  - Reference to online documentation
- Examples
  - Collection of CCSv3 DSP projects and associated PRU code
- Host
  - Common: rCSL, PRU APIs, various helper functions used by examples
  - DSP: CCSv3 loader examples for C674x DSP core

# Package Contents, cont'd

- All projects are for DSP core and are for CCS v3.3
- Most up to date documentation will be found online on TI MediaWiki
- Packaged installers provided for Windows and Linux OSes

# Loading and Running PRU code

# Loading and Running PRU Code

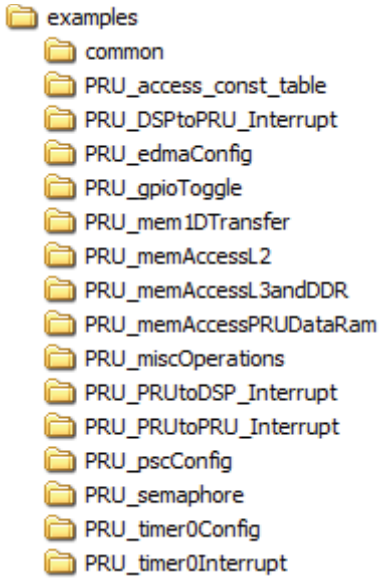
- Host processor of SoC must load code to a PRU and kick off its execution
- Software release contains simple APIs built on top of register layer CSL (included in package).
  - PRU_disable() – Put PRUs into reset state and disable PRUSS via PSC0
  - PRU_enable() – Enable PRUSS via PSC0 and put PRUs into reset state
  - PRU_load() – Enable PRUSS if not enabled, then copy code to IRAM of specified PRU core
  - PRU_run() – Start execution of specified PRU core.
  - PRU_waitForHalt() – Wait for specified PRU to halt, with optional timeout

# Loading Examples

- Software package includes two loading examples
  - DSP loading the PRU using an embedded C array (same way the collection of examples do)
  - DSP loading the PRU using fileio to read a binary file from the hard disk
- Loading examples exercise the PRU APIs
- Located in host/dsp directory of software package

# Examples provided in software package

# Development Examples




- Collection of various examples
  - Show host processor interacting with CPU
  - Show example syntax for PASM assembly code
  - Show how to use PRU constant table for memory access and peripheral configuration
  - Show PRU responding to and generating system events
  - Show the two PRU cores interacting with each other

# Program structure and conventions

- All PRU assembly code files are named with extension .p
- Header files with global macro/struct definitions and #define and .assign statements are named with extension .hp
- All examples include at a minimum
  - c-file: C code that runs on DSP
  - p-file: Assembly code that runs on the PRU
  - hp-file: Header file for PRU assembly code
  - pjt-file: CCS project file
    - Contains prebuild commands to run the PASM tool on the p-file
    - The generated C array file is included in the DSP c-file for loading to the PRU via PRU_load() function