Building Blocks for PRU Development

Embedded Processing
Agenda

• PRU Hardware Overview
• PRU Firmware Development
• Linux Drivers Introduction
PRU Hardware Overview

Building Blocks for PRU Development
ARM SoC Architecture

- **L1 D/I caches**: Single-cycle access
- **L2 cache**: Minimum latency of 8 cycles
- **Access to on-chip SRAM**: 20 cycles
- **Access to shared memory over L3 Interconnect**: 40 cycles
ARM + PRU SoC Architecture

ARM Subsystem
- Cortex-A
- L1 Instruction Cache
- L1 Data Cache
- L2 Data Cache
- On-chip SRAM

Programmable Real-Time Unit (PRU) Subsystem
- PRU0 (200MHz)
  - Shared RAM
  - Inst. RAM
  - Data RAM
- PRU1 (200MHz)
  - Inst. RAM
  - Data RAM

Interconnect
- INTC
- Peripherals

Access Times:
- Instruction RAM = 1 cycle
- DRAM = 3 cycles
- Shared DRAM = 3 cycles
Programmable Real-Time Unit (PRU) Subsystem

• Programmable Real-Time Unit (PRU) is a low-latency microcontroller subsystem.

• Two independent PRU execution units:
  – 32-Bit RISC architecture
  – 200MHz; 5ns per instruction
  – Single cycle execution; No pipeline
  – Dedicated instruction and data RAM per core
  – Shared RAM

• Includes Interrupt Controller for system event handling

• Fast I/O interface: Up to 30 inputs and 32 outputs on external pins per PRU unit.
Now let’s go a little deeper...
**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

**Instruction RAM**
- Typical size is a multiple of 4KB (or 1K Instructions)
- Can be updated with PRU reset

**Special Registers (R30 and R31)**
- R30
  - Write: 32 GPO
- R31
  - Read: 30 GPI + 2 Host Int status
  - Write: Generate INTC Event
Fast I/O Interface
Fast I/O Interface

- Reduced latency through direct access to pins:
  - Read or toggle I/O within a single PRU cycle
  - Detect and react to I/O event within two PRU cycles

- Independent general purpose inputs (GPIs) and general purpose outputs (GPOs):
  - PRU R31 directly reads from up to 30 GPI pins.
  - PRU R30 directly writes up to 32 PRU GPOs.

- Configurable I/O modes per PRU core:
  - GP input modes:
    - Direct input
    - 16-bit parallel capture
    - 28-bit shift
  - GP output modes:
    - Direct output
    - Shift out
**GPIO Toggle: Bench Measurements**

**ARM GPIO Toggle:**

```c
int main(){
    // Configure GPIO module, pinmuxing, etc.
    // Toggle system-level GPIO 3.19 from ARM core
    BitToggle(GPIO_INSTANCE_ADDRESS+GPIO_SETDATAOUT,
              GPIO_INSTANCE_ADDRESS+GPIO_CLEARDATAOUT);
    while();
}

unsigned long BitToggle(unsigned long val1, unsigned long val2){
    asm(   
        " mov r2, #0x00080000 " "\n\t"
        "str  r2,[r0]" "\n\t" // Set GPIO 3.19
        "str  r2,[r1]" "\n\t" // Clear GPIO 3.19
    );
    return val1;
}
```

**PRU IO Toggle:**

```
.origin 0
.entrypoint PRU_GPIO_TOGGLE

PRU_GPIO_TOGGLE:
    // Set PRU GPO 5
    SET    R30, R30, 5
    // Clear PRU GPO 5
    CLR    R30, R30, 5
    HALT
```

- ARM GPIO Toggle: ~200ns
- PRU IO Toggle: ~5ns = ~40x Faster
Integrated Peripherals

- Provide reduced PRU read/write access latency compared to external peripherals
- No need for local peripherals to go through external L3 or L4 interconnects
- Can be used by PRU or by the ARM as additional hardware peripherals on the device
- Integrated peripherals:
  - PRU UART
  - PRU eCAP
  - PRU IEP (Timer)
PRU Read Latencies: Local vs Global Memory Map

The PRU directly accessing internal MMRs (Local MMR Access) is faster than going through the L3 interconnects (Global MMR Access).

<table>
<thead>
<tr>
<th></th>
<th>Local MMR Access (PRU cycles @ 200MHz)</th>
<th>Global MMR Access (PRU cycles @ 200MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRU R31 (GPI)</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>PRU CTRL</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>PRU CFG</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>PRU INTC</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>PRU DRAM</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>PRU Shared DRAM</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>PRU ECAP</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>PRU UART</td>
<td>14</td>
<td>46</td>
</tr>
<tr>
<td>PRU IEP</td>
<td>12</td>
<td>44</td>
</tr>
</tbody>
</table>

Note: Latency values listed are “best-case” values.
PRU “Interrupts”

• The PRU does not support asynchronous interrupts:
  – However, specialized h/w and instructions facilitate efficient polling of system events.
  – The PRU-ICSS can also generate interrupts for the ARM, other PRU-ICSS, and sync events for EDMA.

• From UofT CSC469 lecture notes, “Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring.
  – Interrupts win if processor has other work to do and event response time is not critical
  – Polling can be better if processor has to respond to an event ASAP”

• Asynchronous interrupts can introduce jitter in execution time and generally reduce determinism. The PRU is optimized for highly deterministic operation.
<table>
<thead>
<tr>
<th>Features</th>
<th>AM18x/OMAPL138</th>
<th>AM335x</th>
<th>AM437x</th>
<th>AM571x</th>
<th>AM572x (PG1.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PRU core version</strong></td>
<td>PRUSS</td>
<td>PRU-ICSS1</td>
<td>PRU-ICSS1</td>
<td>PRU-ICSS0</td>
<td>2 x PRU-ICSS</td>
</tr>
<tr>
<td>Number of PRU cores (per subsystem)</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Max frequency</td>
<td>CPU freq / 2</td>
<td>200 MHz</td>
<td>200 MHz</td>
<td>200 MHz</td>
<td>200 MHz</td>
</tr>
<tr>
<td>IRAM size (per PRU core)</td>
<td>4 KB</td>
<td>8 KB</td>
<td>12 KB</td>
<td>4 KB</td>
<td>12 KB</td>
</tr>
<tr>
<td>DRAM size (per PRU core)</td>
<td>512 B</td>
<td>8 KB</td>
<td>8 KB</td>
<td>4 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Shared DRAM size (per subsystem)</td>
<td>--</td>
<td>12 KB</td>
<td>32 KB</td>
<td>--</td>
<td>32KB</td>
</tr>
<tr>
<td>General purpose input (per PRU core)</td>
<td>Direct</td>
<td>Direct; or 16-bit parallel capture; or 28-bit shift</td>
<td>Direct; or 16-bit parallel capture; or 28-bit shift; or 3ch EnDat 2.2; or 9ch Sigma Delta</td>
<td>Direct; or 16-bit parallel capture; or 28-bit shift; or 3ch EnDat 2.2; or 9ch Sigma Delta</td>
<td>Direct; or 16-bit parallel capture; or 28-bit shift</td>
</tr>
<tr>
<td>General purpose output (per PRU core)</td>
<td>Direct</td>
<td>Direct; or Shift out</td>
<td>Direct; or Shift out</td>
<td>Direct; or Shift out</td>
<td>Direct; or Shift out</td>
</tr>
<tr>
<td>GPI Pins (PRU0, PRU1)</td>
<td>30, 30</td>
<td>17, 17</td>
<td>13, 0</td>
<td>20, 20</td>
<td>21*, 21</td>
</tr>
<tr>
<td>GPO Pins (PRU0, PRU1)</td>
<td>32, 32</td>
<td>16, 16</td>
<td>12, 0</td>
<td>20, 20</td>
<td>21*, 21</td>
</tr>
<tr>
<td>MPY/MAC</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Scratchpad</td>
<td>N</td>
<td>Y (3 banks)</td>
<td>Y (3 banks)</td>
<td>N</td>
<td>Y (3 banks)</td>
</tr>
<tr>
<td>CRC16/32</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INTC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Peripherals</td>
<td>n/a</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>UART</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>eCAP</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>no connect</td>
<td>1</td>
</tr>
<tr>
<td>IEP</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>no connect</td>
<td>1</td>
</tr>
<tr>
<td>MII_RT</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>no connect</td>
<td>2</td>
</tr>
<tr>
<td>MDIO</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>no connect</td>
<td>1</td>
</tr>
<tr>
<td>Simultaneous protocols</td>
<td>1</td>
<td>1</td>
<td>2**</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Examples of how people have used the PRU...
Use Case Examples

• Industrial Protocols
  • ASRC
  • 10/100 Switch
• Smart Card
• DSP-like functions
  • Filtering
  • FSK Modulation
• LCD I/F
• Camera I/F
  • RS-485
  • UART
  • SPI
• Monitor Sensors
  • I2C
  • Bit banging
• Custom/Complex PWM
• Stepper motor control

Not all use cases are feasible on PRU
- Development complexity
- Technical constraints
  (i.e. running Linux on PRU)

Development Complexity
PRU Firmware Development

Building Blocks for PRU Development
TI PRU Code Generation Tools (CGT): C Compiler
C Compiler

- Developed and maintained by TI CGT team; Remains very similar to other TI compilers
- Full support of C/C++
- Adds PRU-specific functionality:
  - Can take advantage of PRU architectural features automatically
  - Contains several intrinsics: A list can be found in Compiler documentation
- Full instruction-set assembler for hand-tuned routines

TI PRU CGT Assembly vs C

• Advantages of coding in Assembly over C:
  – Code can be tweaked to save every last cycle and byte of RAM
  – No need to rely on the compiler to make code deterministic
  – Easily make use of scratchpad

• Advantages of coding in C over Assembly:
  – More code reusability
  – Can directly leverage kernel headers for interaction with kernel drivers
  – Optimizer is extremely intelligent at optimizing routines
    • “Accelerating” math via MAC unit, implementing LOOP instruction, etc.
  – Not mutually exclusive; Inline Assembly can be easily added to a C project
PRU Register Header Files
PRU Register Headers

- Created to make accessing a register easier: Register names match those in documentation
- Code Completion feature in CCS automatically lists all members
- Developed to allow a user to program at the register-level or at a bit-field level
  - Note that bit-field accesses could potentially cause some issues with other C compilers (e.g., gcc), but register-level should not.
- PRU cregister mechanism used to leverage constants table when possible
- Currently provides definitions for the following:

  - PRU INTC
  - PRU Config
  - PRU IEP
  - PRU Control
  - PRU ECAP
  - PRU UART
**PRU Register Headers Layout**

```c
/* PRU_CFG_SYSCFG register bit field */
union {
  volatile uint32_t SYSCFG;
  volatile struct{
    unsigned IDLE_MODE : 2;
    unsigned STANDBY MODE : 2;
    unsigned STANDBY_INIT : 1;
    unsigned SUB_MWAIT : 1;
    unsigned rsvd6 : 26;
  } SYSCFG_bit;
} ; // 0x4
```

- **Excerpt from pru_cfg.h**
  - Access register directly `CT_CFG.SYSCFG`
  - Or access specific bitfields
    `CT_CFG.SYSCFG_bit.STANDBY_INIT`

- **Example of how to use in C file**
  - `#include the specific header`
  - Map the constant table entry to register structures
  - Access registers or fields

```c
#include <stdint.h>
#include <pru_cfg.h>

/* Mapping Constant table register to variable */
volatile pruCfg CT_CFG __attribute__((cregister("PRU_CFG",near), peripheral));

/* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
```
Development and Debug Options
Development Within CCS

• In CCS
  – Download and install PRU CGT package via App Center.
  – Open or create new PRU projects just like with any other device.
  – Code completion helps make register accesses easier.

• The Downside
  – It is more difficult to debug while Linux kernel and user application is also running concurrently.
Development Outside of CCS

- **Outside of CCS**
  - Code in your favorite text editor, build via command line
    - Linux and Windows packages available
  - May be easier to script/automate different processes (build or otherwise)

- **The Downside**
  - Can be difficult to debug PRU code
  - Lacks code completion
Debug

• In CCS
  – Easy to view register and variable contents
  – Access to breakpoints and simply stepping mechanism

• Outside CCS
  – Minimal debug control, but some debugfs control provided through remoteproc
  – Start, halt, single-stepping is all console-based
    • Clunky when done by hand, but can potentially be scripted
Linux Drivers Introduction

Building Blocks for PRU Development
ARM + PRU SoC Software Architecture
What Do We Need Linux to Do?

• Load the Firmware
• Manage resources (memory, CPU, etc.)
• Control execution (start, stop, etc.)
• Send/receive messages to share data AND
• Synchronize through events (interrupts)
• These services are provided through a combination of remoteproc/rpmsg + virtio transport frameworks
For More Information

• Visit the PRU-ICSS Wiki: http://processors.wiki.ti.com/index.php/PRU-ICSS

• Download the PRU tools:
  – PRU CGT (Code Gen Tools):
  – Linux drivers for interfacing with PRU:

• Order the PRU Cape: http://www.ti.com/tool/PRUCAPE

• For questions about this training, refer to the E2E Sitara Processors Forum:
  https://e2e.ti.com/support/arm/sitara_arm