ECE 697J – Advanced Topics in Computer Networks

Embedded Control Processor

11/04/03
Overview

• More details on control processor (StrongARM)
  – Overall architecture
  – Typical functions
  – Processor features

• Microengines
  – Architecture and features
  – Differences to conventional processors
  – Pipelining and multi-threading
Purpose of Control Processor

- Functions typically executed by embedded control proc:
  - Bootstrapping
  - Exception handling
  - Higher-layer protocol processing
  - Interactive debugging
  - Diagnostics and logging
  - Memory allocation
  - Application programs (if needed)
  - User interface and/or interface to the GPP
  - Control of packet processors
  - Other administrative functions
System-level View

- Embedded processor can control one or multiple interfaces:
StrongARM Architecture

• ARM V4 architecture with:
  – Reduced Instruction Set Computer (RISC)
  – Thirty-two bit arithmetic with configurable endianness
  – Vector floating point provided via coprocessor
  – Byte addressable memory
  – Virtual memory support
  – Built-in serial port
  – Facilities for kernelized operating system
StrongARM Memory Architecture

- Memory architecture
  - Uses 32-bit linear address space
  - Byte addressable

- Memory Mapping
  - Allocation of address space to different system components
  - Access to memory is translated into access to component
  - Needs to be carefully crafted

- StrongARM assumes byte addressable memory
  - Underlying memory uses different size (SDRAM)
  - How does this work?

- Support for Virtual Memory
  - For demand paging to secondary storage
StrongARM Memory Map

<table>
<thead>
<tr>
<th>Contents</th>
<th>SDRAM Bus:</th>
<th>Scratchpad</th>
<th>Microengine xfer</th>
<th>Microengine CSR's</th>
<th>AMBA xfer</th>
<th>Reserved</th>
<th>System regs</th>
<th>Reserved</th>
<th>PCI Bus:</th>
<th>PCI memory</th>
<th>PCI config</th>
<th>Local PCI config</th>
<th>SRAM Bus:</th>
<th>SlowPort</th>
<th>SRAM CSR's</th>
<th>Push/Pop cmd</th>
<th>Locks</th>
<th>BootROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>FFFF</td>
<td>FFFF</td>
<td>C000</td>
<td>B000</td>
<td>A000</td>
<td>9000</td>
<td>8000</td>
<td>4000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
• Memory is shared between StrongARM and Microengines
• Same data, but different addresses
• What impact does this have?
  – Pointers need to be translated
  – Data structures with pointers cannot be shared. Why?
**StrongARM Peripherals**

- Peripherals on StrongARM:
- UART
- Four 24-bit countdown timers
  - Can be configured to 1, 1/16, 1/256 of StrongARM clock
- Four general purpose pins
  - For special off-chip devices
- One real-time clock
  - Tick per second
- Clock is for large granularity timing (e.g., route aging), counters are for small granularity
StrongARM Misc

• StrongARM can support kernelized OS
  – Kernel at highest priority
  – Kernel controls I/O and devices
  – User-level processes with lower privileges

• Coprocessor 15
  – MMU configuration
  – Breakpoints for testing

• Summary
  – StrongARM is full-blown processor with powerful and general features
Microengines

- Microengines are data-path processors of IXP1200
- IPX1200 has 6 microengines
- Simpler than StrongARM
- A bit more complex to use
- Often abbreviated as uE
Microengine Functions

• uEs handle ingress and egress packet processing:
  – Packet ingress from physical layer hardware
  – Checksum verification
  – Header processing and classification
  – Packet buffering in memory
  – Table lookup and forwarding
  – Header modification
  – Checksum computation
  – Packet egress to physical layer hardware
Microengine Architecture

• uE characteristics:
  – Programmable microcontroller
  – RISC design
  – 128 general-purpose registers
  – 128 transfer registers
  – Hardware support for 4 threads and context switching
  – Five-stage execution pipeline
  – Control of an Arithmetic and Logic Unit
  – Direct access to various functional units
uE as Microsequencer

- Microsequencer does not contain native operations
  - Control unit is much “simpler”
- Instead of using instructions, uE invokes functional units
- Example 1:
  - uE does not have ADD R2,R3 instruction
  - Instead: ALU ADD R2, R3
  - “ALU” indicates that ALU should be used
  - “ADD” is a parameter to ALU
- Example 2:
  - Memory access not by simple LOAD R2, 0xdeadbeef
  - Instead: SRAM LOAD R2, 0xdeadbeef
- Altogether similar to normal processor, but more basic
# Microengine Instruction Set (1)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic, Rotate, And Shift Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>Perform an arithmetic operation</td>
</tr>
<tr>
<td>ALU_SHF</td>
<td>Perform an arithmetic operation and shift</td>
</tr>
<tr>
<td>DBL_SHIFT</td>
<td>Concatenate and shift two longwords</td>
</tr>
<tr>
<td><strong>Branch and Jump Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>BR, BR=0, BR!=0, BR&gt;0, BR&gt;=0, BR&lt;0, BR&lt;=0, BR=count, BR!=count</td>
<td>Branch or branch conditional</td>
</tr>
<tr>
<td>BR_BSET, BR_BCLR</td>
<td>Branch if bit set or clear</td>
</tr>
<tr>
<td>BR=BYTE, BR!=BYTE</td>
<td>Branch if byte equal or not equal</td>
</tr>
<tr>
<td>BR=CTX, BR!=CTX</td>
<td>Branch on current context</td>
</tr>
<tr>
<td>BR_INP_STATE</td>
<td>Branch on event state</td>
</tr>
<tr>
<td>BR.Signal</td>
<td>Branch if signal deasserted</td>
</tr>
<tr>
<td>JUMP</td>
<td>Jump to label</td>
</tr>
<tr>
<td>RTN</td>
<td>Return from branch or jump</td>
</tr>
</tbody>
</table>
## Microengine Instruction Set (2)

<table>
<thead>
<tr>
<th>CSR</th>
<th>CSR reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST_WR</td>
<td>Write immediate data to thd_done CSRs</td>
</tr>
<tr>
<td>LOCAL_CSR_RD, LOCAL_CSR_WR</td>
<td>Read and write CSRs</td>
</tr>
<tr>
<td>R_FIFO_RD</td>
<td>Read the receive FIFO</td>
</tr>
<tr>
<td>PCI_DMA</td>
<td>Issue a request on the PCI bus</td>
</tr>
<tr>
<td>SCRATCH</td>
<td>Scratchpad memory request</td>
</tr>
<tr>
<td>SDRAM</td>
<td>SDRAM reference</td>
</tr>
<tr>
<td>SRAM</td>
<td>SRAM reference</td>
</tr>
<tr>
<td>T_FIFO_WR</td>
<td>Write to transmit FIFO</td>
</tr>
</tbody>
</table>

- CSR = Control and Status Register
**Microengine Instruction Set (3)**

<table>
<thead>
<tr>
<th>Local Register Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND_BST, FIND_BSET_WITH_MASK</td>
<td>Find first 1 bit in a value</td>
</tr>
<tr>
<td>IMMED</td>
<td>Load immediate value and sign extend</td>
</tr>
<tr>
<td>IMMED_B0, IMMED_B1, IMMED_B2, IMMED_B3</td>
<td>Load immediate byte to a field</td>
</tr>
<tr>
<td>IMMED_W0, IMMED_W1</td>
<td>Load immediate word to a field</td>
</tr>
<tr>
<td>LD_FIELD, LD_FIELD_W_CLR</td>
<td>Load byte(s) into specified field(s)</td>
</tr>
<tr>
<td>LOAD_ADDR</td>
<td>Load instruction address</td>
</tr>
<tr>
<td>LOAD_BSET_RESULT1, LOAD_BSET_RESULT2</td>
<td>Load the result of find_bset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTX_ARB</td>
<td>Perform context swap and wake on event</td>
</tr>
<tr>
<td>NOP</td>
<td>Skip to next instruction</td>
</tr>
<tr>
<td>HASH1_48, HASH2_48, HASH3_48</td>
<td>Perform 48-bit hash function 1, 2, or 3</td>
</tr>
<tr>
<td>HASH1_64, HASH2_64, HASH3_64</td>
<td>Perform 64-bit hash function 1, 2, or 3</td>
</tr>
</tbody>
</table>
Microengine Memories

- uEs views memories separately
  - Not one address space like StrongARM
- Requires programmer to decide on memories to use
  - Different memories require different instructions
- Also: instruction store is in different memory than data
  - Not a van-Neumann/Princeton architecture...
Execution Pipeline

- uEs have five-stage pipeline:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fetch the next instruction</td>
</tr>
<tr>
<td>2</td>
<td>Decode the instruction and get register address(es)</td>
</tr>
<tr>
<td>3</td>
<td>Extract the operands from registers</td>
</tr>
<tr>
<td>4</td>
<td>Perform ALU, shift, or compare operations and set the condition codes</td>
</tr>
<tr>
<td>5</td>
<td>Write the results to the destination register</td>
</tr>
</tbody>
</table>

- In proper pipeline operation, one instruction is executed per cycle
# Pipelining

<table>
<thead>
<tr>
<th>Time</th>
<th>clock</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4</th>
<th>stage 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>inst. 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>inst. 2</td>
<td>inst. 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>inst. 3</td>
<td>inst. 2</td>
<td>inst. 1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>inst. 4</td>
<td>inst. 3</td>
<td>inst. 2</td>
<td>inst. 1</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>inst. 5</td>
<td>inst. 4</td>
<td>inst. 3</td>
<td>inst. 2</td>
<td>inst. 1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>inst. 6</td>
<td>inst. 5</td>
<td>inst. 4</td>
<td>inst. 3</td>
<td>inst. 2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>inst. 7</td>
<td>inst. 6</td>
<td>inst. 5</td>
<td>inst. 4</td>
<td>inst. 3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>inst. 8</td>
<td>inst. 7</td>
<td>inst. 6</td>
<td>inst. 5</td>
<td>inst. 4</td>
</tr>
</tbody>
</table>
Pipelining Problems

• What can lead to cases where pipeline does not operate as desired?
  – Data dependencies
  – Control dependencies
  – Memory accesses
• What happens in either case?
• How can these cases be made less frequent?
• How can the impact be reduced?
Pipeline Stalls

- K: ADD R2, R1, R2
- K+1: ADD R3, R2, R3

<table>
<thead>
<tr>
<th>clock</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4</th>
<th>stage 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>inst. K+3</td>
<td>inst. K+2</td>
<td>inst. K+1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>inst. K+4</td>
<td>inst. K+3</td>
<td>inst. K+2</td>
<td>inst. K+1</td>
<td>-</td>
</tr>
</tbody>
</table>

- Control dependencies, memory have even bigger impact
Hardware Threads

• uEs support four hardware thread contexts
  – One thread can execute at any given time
  – When stall occurs, uE can switch to other thread (if not stalled)

• Very low overhead for context switch
  – “Zero-cycle context switch”
  – Effectively can take around three cycles due to pipeline flush

• Switching rules
  – If thread stalls, check if next is ready for processing
  – Keep trying until ready thread is found
  – If none is available, stall uE and wait for any thread to unblock

• Improves overall throughput
• Side note: why not have 24 uEs with 1 thread?
Threading Illustration

- **Thread 1**: Time $t_1$.
- **Thread 2**: Time $t_2$.
- **Thread 3**: Time $t_3$.

- **Context Switch**: Arrows indicating context switch between threads.

- **Time**: The timeline for the threads.
• “Random” RISC processor (MIPS R7000)
• 300 MHz, 16k/16k caches, .25 um, 1997
• Memory takes most area
Next Class

• Continue with Microengines
  – Instruction store, hardware registers
  – FBI and FIFO
  – Hash unit
• SDK
• Read chapters 20 & 21