ARM 9 Instruction Set Architecture Introduction with Performance Perspective

Joe-Ming Cheng, Ph.D.

ARM-family processors are positioned among the leaders in key embedded applications. Many presentations and short lectures have already addressed the ARM's applications and capabilities. In this introduction, we intend to discuss the ARM's instruction set uniqueness from the performance prospective. This introduction is also trying to follow the approaches established by two outstanding textbooks of David Patterson and John Hennessey [PetHen00] [HenPet02].

1.0 ARM Instruction Set Architecture

Processor *instruction set architecture* (***ISA***) choices have evolved from accumulator, stack, register-to-memory, to register-register (load-store) organization. ARM 9 ISA is a load-store machine. ARM 9 ISA takes advantage of its smaller set of registers (16 vs. many 32-register processors) to incorporate more direct controls and achieve high encoding density. ARM's load or store multiple register instruction, for example, allows enlisting of all possible registers and conditional execution in one instruction.

The Thumb mode instruction set is another example of how ARM ISA facilitates higher encode density. Rather than compressing the code, Thumb-mode instructions are two 16-bit instructions packed in a 32-bit ARM-mode instruction space. The Thumb-mode instructions are a subset of ARM instructions. When executing in Thumb mode, a single 32-bit instruction fetch cycle effectively brings in two instructions. Thumb code reduces access bandwidth, code size, and improves instruction cache hit rate.

Another way ARM achieves cycle time reduction is by using Harvard architecture. The architecture facilitates independent data and instruction buses. Consider an instruction mix with 30% load/store instructions. To ensure that the pipeline running nearly stall-free, 1.3 average memory accesses is needed for every instruction execution. The Harvard architecture allows the peak memory bandwidth to be 2.0 instead of 1.0. In addition, the independent data and address buses increase the percentage of sequential memory access. At present, ARM9 non-sequential access could take one or more extra cycles.

This proceeding paper provides a brief overview of the following subjects with emphasis on running-time performance:

- ARM 922 programmer's model – architect
- Brief performance introduction on throughput, response time, and workload
- ARM assembly code for C-Programming "if..then..else" and "while" loop; possible use of loop-unrolling, strength reduction for faster execution
- Built-in semaphore instruction, "SWP" and example of creating "Mutex" semaphore, and semaphore operations of "take" and "give" with SWP
- ARM and Thumb mode transition
- Interrupt handling and code example
- Basic caching considerations

1.1 ARM9 Programmer's Model

The ARM processor has a total of 37 registers in seven modes. User, System, and Fast Interrupt are ARM9 processor modes. Each mode sees R0..R15 general-purpose registers, where R13, R14, and R14 serve as stack pointer, link register, and program counter. Some registers are shared among several modes. The current program status register (CPSR, not shown) and saved program status register (SPSR, not shown) retain arithmetic flags, Thumb/ARM mode, interrupt, and processor mode status information. Figure 1 shows ARM's instruction set encoding format. It includes generic RISC format; unique conditional executable field; and register list for individual register selection.

Table 1: ARM Processor Modes						
	Privileged Modes					
		Exception Mode				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fig
R10	R10	R10	R10	R10	R10	R10_fig
R11	R11	R11	R11	R11	R11	R11_fig
R12	R12	R12	R12	R12	R12	R12_fig
R13 – SP	R13 – SP	R13_SVC	R13_abt	R13_und	R13_inq	R13_fig
R14 - Link	R14 - Link	R14_SVC	R14_abt	R14_und	R14_inq	R14_fig
R15 - PC	pc	pc	pc	pc	pc	pc

unique

generic

unique

Instruction (bits 31 … 0)													
Data processing immediate shift	cond [1]	0 0 0	opcode	S	Rn	Rd	shift amount	shift	0	Rm			
Miscellaneous instructions: See Figure 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x x x x x x x x x x x x	0	x x x x						
Data processing register shift [2]	cond [1]	0 0 0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm		
Miscellaneous instructions: See Figure 3-1	cond [1]	0 0 0	1 0 x x	0	x x x x x x x x x x x x	0	x x	1	x x x x				
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0 0 0	x x x x x x x x x x x x x x x x x	1	x x	1	x x x x						
Data processing immediate [2]	cond [1]	0 0 1	opcode	S	Rn	Rd	rotate	immediate					
Undefined instruction [3]	cond [1]	0 0 1	1 0	x	0 0	x x x x x x x x x x x x x x x x x x x x							
Move immediate to status register	cond [1]	0 0 1	1 0	R	1 0	Mask	SBO	rotate	immediate				
Load/store immediate offset	cond [1]	0 1 0	P	U	B	W	L	Rn	Rd	immediate			
Load/store register offset	cond [1]	0 1 1	P	U	B	W	L	Rn	Rd	shift amount	shift	0	Rm
Undefined instruction	cond [1]	0 1 1	x x x x x x x x x x x x x x x x x x x x	1	x x x x								
Undefined instruction [4,7]	1 1 1 1	0	x x x x x x x x x x x x x x x x x x x x x x x x x x x										
Load/store multiple	cond [1]	1 0 0	P	U	S	W	L	Rn	register list				
Undefined instruction [4]	1 1 1 1	1 0 0	x x x x x x x x x x x x x x x x x x x x x x x x x										
Branch and branch with link	cond [1]	1 0 1	L	24-bit offset									
Branch and branch with link and change to Thumb [4]	1 1 1 1	1 0 1	H	24-bit offset									
Coprocessor load/store and double register transfers [6]	cond [5]	1 1 0	P	U	N	W	L	Rn	CRd	cp_num	8-bit offset		
Coprocessor data processing	cond [5]	1 1 1 0	opcode1	CRn	CRd	cp_num	opcode2	0	CRm				
Coprocessor register transfers	cond [5]	1 1 1 0	opcode1	L	CRn	Rd	cp_num	opcode2	1	CRm			

Figure 1 ARM's instruction set encoding format includes generic RISC format; unique conditional executable field; and register list for individual register selection.

2.0 Performance and Workloads

Response time and throughput are two primary performance measurements. **Response Time,** Figure 3, measures the time between input (stimulus) and output (response). **Throughput**, Figure 4, measures the number of inputs that can be processed per unit time (indication of system capacity). A system has many bottlenecks. Multiple test cases or test suites, called **workloads**, are usually used to expose various bottlenecks and to gauge system performance.

When performance hotspots are identified, the iterative tuning process will take place to enhance performance. Many texts addressed various application, OS, IO, and DB tunings through system configuration/setup, network topology, and code tuning. Performance tuning is not addressed here. In general, improving response time usually is harder than improving throughput.

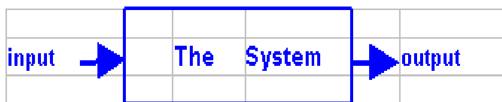

Figure 2 System with input and output

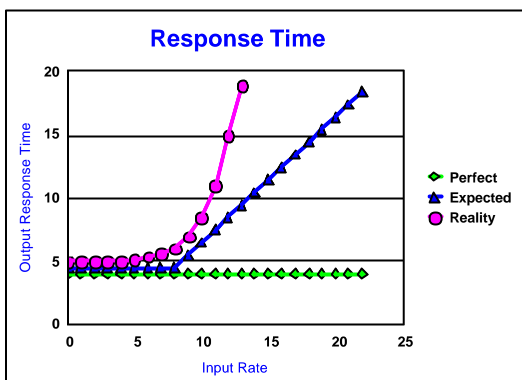

Figure 3 Response time vs. input rate

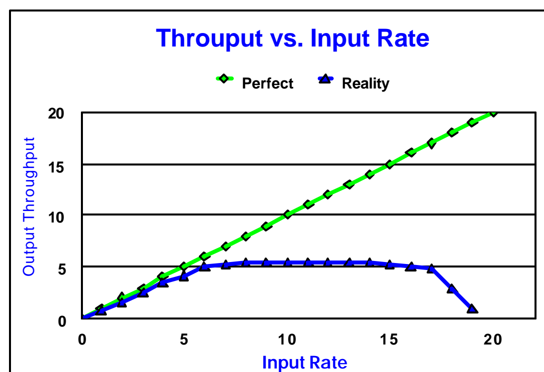

Figure 4 Throughput vs. input rate

2.1 Benchmarks

Benchmark suites are workloads defined publicly. The SPEC benchmark suites include many applications: Boolean function optimization, LISP interpretation, SPICE circuit simulation, and others. DLX instruction set includes recent instruction features from MIPS, Power PC, Precision Architecture, and SPARC. Five benchmark results for DLX are charted in Figure 5. The execution traces showed there were over 25% load/store executions, and over 17% of branch executions. An ISA (instruction set architecture) and machine organization usually need to adequately address these two aspects.

2.2 Application Code Footprint

Embedded applications usually have Read-Only or flash memory for storing program and data. The size of storage directly affects the cost. The run-time storage size is often referred to as **footprint**. An application program footprint may reside on SRAM, DRAM, and secondary storage. Frequent accesses to DRAM and secondary storage may result in severe performance degradation. SRAM, usually implemented with 6-transistors cell, has size limits. One-transistor DRAM or secondary storage could post access time issues and may need buffers (such as cache) for speeding up the effective access. ARM 920/922, 946 and other processor versions have cache provisions to improve effective access.

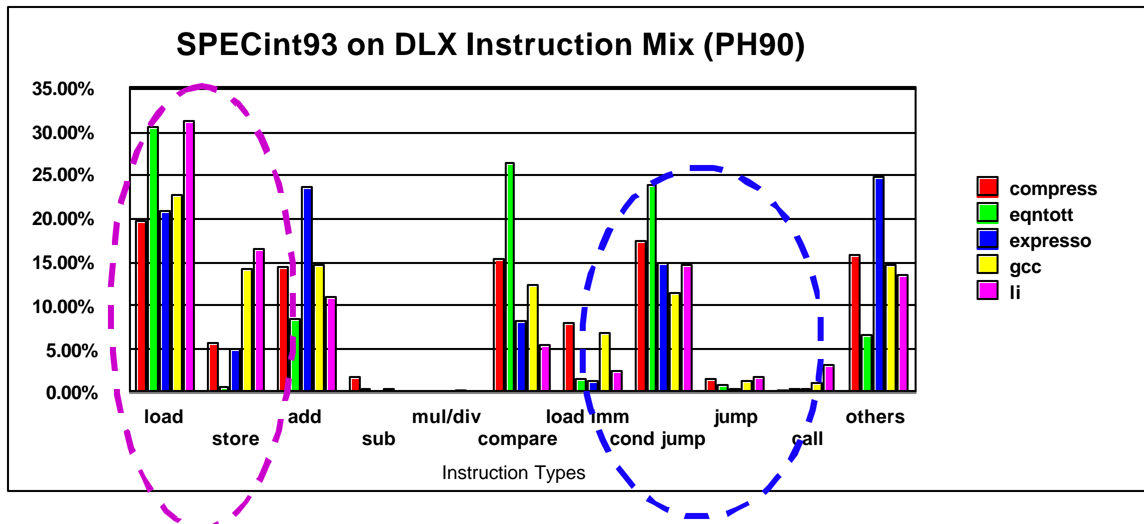

Figure 5 SPECinte94 DLX instruction mix for compress, eqntott, expresso, gcc, and li benchmarks

3.0 C and C++ Programming Performance Implications

The time required to execute an application is the product of the number of instructions (path length), the number of cycles per instruction, and the cycle time. Solutions for shorter execution time and smaller code size or foot print often present various challenges to embedded application developers. Many applications are now written in C/C++ or other high-level languages. To write execution-efficient code requires intimate awareness of path length (cost) of compiler generated codes on key structures such as: functional call entry and exit, if-then-else, switch, for, while structures [IBM96]. OS kernel components and performance critical routine may need to be written in assembly code. Strict adherence to compiler vendor or Arm Procedure Call Interface standard is crucial to ensure the inter-operation between compiler-generated codes and the assembly codes created manually.

3.1 Function or Procedure Call

Function (or procedure) allows grouping of commonly used codes as a unit that can be used repeatedly. In many applications, over 90% of execution time dwells on less than 10% functions. These so-called hot spots, usually ranked by 1st, 2nd, 3rd, based on overall execution time taken, may be identified by software tools. For the hot-spots identified, there are cases that the function entry and exit overheads present significant execution time, such as 20% to 80% of the execution.

STMFD and LDMFD (and their variants) are single instructions that allow save and restore ARM registers.

STMFD Store multiple registers & R14 (Link register, stores the return address) to the stack (pointed by R13) on subroutine entry
xmp: STMFD R13!,{<registers>,R14}

LDMFD load (or restore) multiple registers and PC from the stack (pointed by R13) and returning to caller
xmp: LDMFD R13!,{<registers>,PC}

The instruction execution trace in Figure 6 shows a function that takes 25 cycles to enter and exit. The function body takes only 20 cycles to execute.

Instruction Execution Trace	Cycles
0x0001C1E8 BLX 0x1E88	2
BRANCH: from 0x0001C1E8 --> 0x00001E88 due to PC CHANGE	
>> void GameStart::CheckMode(Mode &FastAttach)	10
0x00001E88 STMFD r13!,{r2-r6,r14}	
STORE: 0x04002F58 = 0x00000004	
STORE: 0x04002F5C = 0x00000030	
STORE: 0x04002F60 = 0x040075E4	
STORE: 0x04002F64 = 0x04007784	
STORE: 0x04002F68 = 0x3010602C	
STORE: 0x04002F6C = 0x0001C1ED	
0x00001E8C MOV r4,r0	1
0x00001E90 MOV r0,#0x80	1
0x00001E94 MOV r5,r1	2
… other codes	16
LDMFD r13!,{r2-r6,pc}	15

Figure 6 A procedure call execution trace

It is not uncommon to see often used procedures where the body consumed less than 50% of the execution time. C++ codes often have many functions with a small body such as an accessor. These functions could be the sources of major overheads if used often. There are three basic remedies to mitigate the procedure overheads:

1. Procedure in-lining
2. Increase the size of procedure body
3. Reduce the number of procedure calls

C++ Accessor ensures better protection.
Example:

```
class stack {        // Prof. Ira Phol
private:
        char s[max_len];
        int top;
        enum {EMPTY = -1, FULL = max_len-1};
public:
        void rest(){top=EMPTY;}
        void push(char c){top++; s[top]=c;}
        char pop() { return (s[top--]); }
        char top_of(){return s[top]; }
        int get_top(){return top; }
        boolean empty(){ return (boolean)(top == EMPTY); }
        boolean full(){  return (Boolean)(top== FULL);  }
} ;
```

Figure 7 Accessor function code example

Reducing procedure entry and exit overhead allows programs to run faster. Likewise, when control loop overhead becomes excessive, loop-unrolling may be considered to reduce overhead. Extended use of speed-up techniques often leads to a less desired programming style. A classical remedy to this problem is good documentation.

3.2 Indirection

Indirection often incurs additional machine cycles. Pointer, array, and object local accesses involve indirection. Many high-level programming languages have provisions for handling programming life cycle. The uses of access control, virtual function, and inheritance involve indirections. In addition to extra machine cycles, indirection decreases the caching efficiency. The long cache miss memory access could introduce major performance degradation for systems with processor cache.

Many developers oppose using C++ for writing embedded applications. Some projects use C++ object just for grouping functions and datum in a secured way, but not using the inheritance, polymorphism. There are large projects where the full embedded programs are written in assembly code. Code size, run-time footprint, response time, power, and cost are important factors of embedded system architecture.

3.3 If then else

Conditional branching statements in C including if-then-else or if-then-else-if is often implemented using conditional branch machine instruction. Method#1 is a common structure used for assembly code generation. The “bnq”, branch not equal instruction, machine instruction is used for choosing the execution of action 1 or action 2. The 32-bit ARM instruction also supports the use of compilation structure Method#2. The “bnq” is not needed for this method. Rather, every instruction in action 1 body is an **conditional-executed instruction**. When path length (number of instructions) of action 1 is short, for example, four or less, Method#1 can run 7~10% faster than Method#1.

Table 2	Two compiler code generation choices
C code	if (x+y) ==0 .. action 1 else .. action 2
Assembly code Method #1 Typical code generation	Assembly code generation method #2 ARM Support:
• add • compare • bnq b: • a: .. action 1 .. Branch • b: .. action 2 .. Branch	• add • compare • (void) • a: .. action 1 .. Branch (conditional executed code) • b: .. action 2 .. Branch

Comparison of method#1 and method #2

Consider BNE instruction takes extra 4 cycles when branch is taken, for code generation Method#2, Figure 8, and action 1 contains 4 or less instruction, it will save 0,1, or 2 cycles. In the branch not taken case, method#2 saves one execution cycle. For many control applications, an average of one instruction in 6 is a branch. The conditional-executed instruction allows 7~10% cycle reduction, which is significant. The SKIPs in the following trace are conditional-executed instruction turned NO OP.

if..then..else Instruction Trace	Cum. Time	Inst. Time
>> if(Lock.A==MyTask.Key1){ 0x00001EA0 LDR r0,[r5,#4]	5.0ns	5.0ns
0x00001EA4 LDR r2,[r4,#0x160]	10.0ns	5.0ns
LOAD: RTOS.CurrentTask(@0x04005714) = 0x02 0x00001EA8 CMP r0,r2	15.0ns	5.0ns

>> Lock.A.Count++; SKIP: 0x00001EAC LDREQB r0,[r5,#5] SKIP: 0x00001EB0 ADDEQ r0,r0,#1 SKIP: 0x00001EB4 STREQB r0,[r5,#5] SKIP: 0x00001EB8 BEQ 0x1f90		
>> else if(Lock.A==OsTask){ 0x00001EBC CMP r0,#0xff	40.0ns	5.0ns

Figure 8 "if..then..else" compile Method# 2 code trace

Most ARM instructions can be conditionally executed

Figure 9 shows the conditional-executed instruction format. The LDREQD and LDRNE instructions listed below describe the conditions for the instructions to be executed.

- LDREQD // LDR{ }D is executed if "Z" is set, equal
- LDRNE // LRR{ }D is executed if "Z not set", not equal.

31	30	29	28				7	6	5	4	3	2	1	0
	cond													
							Condition							
0	0	0	0	EQ	Equal		Z set							
0	0	0	1	NE	Not equal		Z clear							
0	0	1	0	CS/HS	Carry set		C set							
0	0	1	1	CC/LO	Carry clear		C clear							

Figure 9 Example of coded conditional field

High Instruction Encoding Density

31	28	27	26	25	24	23	22	21	20	19	16	15		0
Cond		1	0	0	P	U	1	0	0	Rn		register list		
	Cond		The condition that allows the instruction to be executed											
	P,U		addressing mode											
	Rn		Base register used; xmp: R13, stack											
	Register List			each register can be independently selected										

Figure 10 Instruction format contains conditional field and register selection list

The ARM load/store multiple format, Figure 10, exemplifies high instruction encoding. The bit 15..0 field allows any combination of registers to be selected. A processor with large number of registers can not encode the register-selection field this way. Then, when loading and storing non-contiguous registers, multiple instructions are needed to load or store each register. The ARM load/store multiple instructions also have the conditional execution field. Smart compiler can use this feature to generate compact fast-running code.

3.4 while loop

In general, codes generated for "while (flag.x != 0) action 1" are:

```
a: cmp r4,#0  // compare
```

```
    bne b:
      .. action 1 ..  branch a: // while core code
b:
```

3.5 Thumb Mode and Transition between ARM and Thumb Mode (51)

The ARM Thumb-mode instructions are encoded in 16 bits. A single 32-bit instruction cycle fetches two Thumb instructions. Code segments generated with Thumb mode instructions have smaller footprints, require less instruction bandwidth, and could improve caching and pipeline efficiencies. Transition from ARM to Thumb mode execution, in some cases, requires additional instructions (Figure 11). These extra instructions are sources of performance and footprint overheads.

Address	Instruction	Comment
	…	//prepare R2 for target address || 0x1
0x00011544	BLX R2	//Branch, Link, and exchange
C: Routine()		

0x00007554
CRoutine:
	push {R4,R5,R7,R14}
	0x04003CE0 = 0x04004074
	0x04003CE4 = 0x04005074
	0x04003CE8 = 0x00000000
	0x04003CEC = **0x00011548**
		// Thumb mode operation
		// push R4
		// push R5
		// push R7
		// push R14

0x00007556	MOV r4,r0	// Thumb mode operation
0x00007558	MOV r0,#5	// Thumb mode operation

Figure 11 Instruction execution trace of ARM to Thumb mode transition

3.6 Build-in Semaphore Support and RTOS

Atomic operation is a collection of execution steps executed as a single indivisible unit (without interruption). **Semaphore** is a condition of events that a task (process) is waiting for. Semaphore is usually used to ensure proper serialization of shared resource. **SWP** is an ARM9 built-in Atomic instruction. SWP can be used to construct mutual-exclusion (mutex) types of Semaphore.

For many applications, the **RTOS** (real-time operating system) is ported from one to another processor. The Atomic operation is emulated through disabling all interrupts instead of using processor built-in Atomic instruction. This approach is general-purpose, but less efficient and is more restricted. Developers should take the advantage of using built-in Atomic or Semaphore instruction.

Figure 12 shows a "Mutex" Semaphore implementation based on the SWP instruction (source: ARM Architecture Reference Manual [Seal00]). The semaphore is a memory location pointed by R0. The SWP instruction allows the exchange of a "looking code = -1" in R2 with memory[R0] in one atomic operation. If the semaphore is currently owned by a process ID, the semaphore is restored. Or, if another process is looking at the semaphore, the process is entering the wait mode and will try to "look" again at a later point. In case the semaphore is "free", the process puts its own ID on the semaphore and assumes the ownership. When the process completes its task, it then needs to relinquish the semaphore (set ID = 0).

Embedded systems usually have basic to extended capabilities of servicing real-time events. Complete real-time OSs, available for purchase, include not only OS kernel, threading, semaphore capabilities but also include application domain functions such as drivers for a complete network protocol stack.

```
ARM Semaphore Instruction Mutex Example (ARM Arch. Ref. Manual)
on entry   R0                        Holds the address of semaphore
           R1                        Holds the Process ID requesting the lock

           mvn                       R2, #0         ; load the looking value of -1 in R2

spinin     swp                       R3, R2, [R0]   ; m[R0] ->R3, m[R0]<-R2
           cmn                       R3,#1          ; anyone else tying to "look"?

           .... OS call to sleep process            ; remember the SKIP when not equal

           beq                       spinin         ; yes, wait for my turn

           cmp                       R3, #0         ; Is lock free
           strne                     R3, [R0]       ; no, restore the lock
           .... OS call to sleep process            ; remember the SKIP when not equal
           bne                       spinin

           str                       R1,[R0]        ; assume lock ownership with own ID

           ... insert your critical here

spinout    swp                       R3, R2, [R0]   ; look at the lock
           cmn                       R3, #1         ; anyone is looking?
           .... OS call to sleep process            ; remember the SKIP when not equal
           beq                       spinout

           cmp                       R3, R1         ; check if I am the owner
           bne                       SemaphoreCorruptedErrorHandler
           mov                       R2, #0
           str                       R2, [R0]       ; free the lock
```

Figure 12 Mutex lock implementation with ARM9 built-in SWP instruction

Basic OS core can be built on interrupt services and semaphore functions. When an exception (event) occurs, the execution is forced to start from a fixed memory address corresponding to the type of exception. Figure 13 shows an ARM FIQ (fast interrupt request) interrupt service handler.

```
0x00012C00   ..........                               //FIQ occurred

0x0000001C   B        0x1b8                           //FIQ  Vector, Critical Interrupt Handler

0x000001B8   STMFD    r13!,{r0-r3,r12,r14}            //Critical Interrupt Handler
     store:  0x04003D18 = 0x00001410                  //r0 saved in stack
     store:  0x04003D1C = 0x00000000                  //r1 saved in stack
     store:  0x04003D20 = 0x00000000                  //r2 saved in stack
     store:  0x04003D24 = 0x00000000                  //r3 saved in stack
     store:  0x04003D28 = 0xFFFFFFFF                  //r12 saved in stack
     store:  0x04003D2C = 0x00012C0C                  //Return Address + 4

             Poll interrupt causes

             LDR      R0, 0x224                       //Load Argument
             BL       0x05b4                          //Branch into interrupt Service routine

0x00005CB4   STMFD    r13!,{r4-r6,r14}                //Interrupt Service Routine
             ..........
             LDMFD    r13!,{r4-r6,r14}

0x000001C4   LDMFD    r13!,{r0-r3,r12,r14}
```

Figure 13 A fast interrupt service routine outline

4.0 Caching and Basic Hardware Consideration

Large amounts of high-speed memory would simplify developers' tasks. However, power and cooling constraints have been issues for mainframe, workstation platforms, and even so for most embedded applications. Memory hierarchy takes advantage of spatial and temporal memory accessing localities and cost/performance of memory technologies. Many CDC models and RCA1800 employed high-speed memory pages along with slower, but cheaper, memory pages. Programmers need to be speed-location aware and to move speed-critical codes to fast memory pages. Alternatively, a more common choice is the use of higher-speed buffer called cache. Cache removes the need of speed-location aware and code overlay.

Modern processor caching structures usually consists of three functional units: instruction/data caching, prefetch buffer, and miss/victim caching [Jouppi90]. Choices of line size and associativity (for caching unit) are usually based on the workload expected. The SPICE bench mark cache simulation results, Figure 14, showed the access overhead (0% means no cache miss) vs. the 1,2,4-way associativity and 16..256byte line sizes. The "A4" denotes the memory access is 4X the processor clock time. B0, B4, and B8 stand for prefetch buffer of line size of 0, 4, and 8. In this chart the optimal cache design point choice is: 4-way set associative, 32-byte line size, and a prefetch buffer of 8 lines

It is indicated that the ARM9TDMI core is not generally available. The necessary cache signal interfaces are not directly accessible. The developer needs to consult ARM or IC Foundries for available cache configurations. The ARM920/922 core supports 64-way associativity [Furber00]. The high degree of associativity reduces conflict miss, which makes the use of miss/victim cache unnecessary. The high-degree cache associativity performs well for multi-threading or multi-tasking applications. In general, the temporal and spatial localities of data caching are not as good as instruction caching. The developer could resort to a larger data cache or devise intelligent prefetch algorithms.

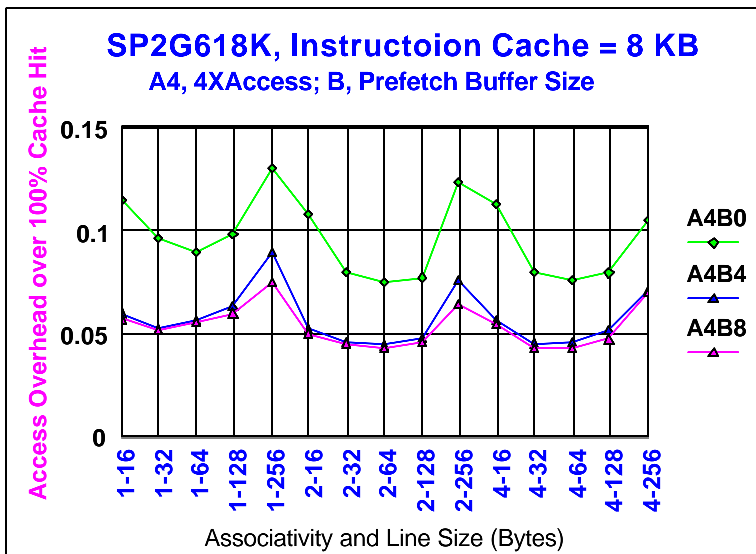

Figure 14 Spice benchmark cache access overhead on 45 cache organization design points

Summary

In summary, ARM instruction set provides high functional density for control and other key applications. Thumb mode instruction allows execution footprint size reduction with slight performance overhead. Successful development of embedded application often requires establishment of clear performance targets and workloads at the start. For codes on the response critical paths, the developer should be aware of the machine code generated by the compiler and be prepared to write fast assembly or C codes if necessary.

References

Furber00 Steve Furber, "ARM System-on-chip Architecture", ISBN 0-201-67519-6, Addison-Wesley

HenPet02 John L. Hennessy and David A. Patterson, "Computer Architecture - A Quantitative Approach", Morgan Kaufmann, SF, CA

IBM96 Hoxey et al edited, "The PowerPC Compiler Writer's Guide", ISBN 0-9649654-0-2, Warthman Associates

Jouppi90 Norm P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", IEEE, 1990

PetHen00 David A. Patterson and John L. Hennessy ,"Computer Organization and Design", Morgan Kaufmann, SF, CA

Seal00 David Seal, "ARM Architecture Reference Manual", ISBN 0-201- 737192, Addison-Wesley

Author Biography

Joe-Ming Cheng is a member of performance analysis team at Hitachi Global Storage Technology and is also a part-time Associate professor at Silicon Valley University. He has worked at IBM Research and Development laboratories for 25 years on algorithm, SOC, storage system, embedded system, and design automation tool developments. He received an Outstanding Innovation Award and an Outstanding Achievement Award from IBM. He has also worked on guidance system and air-borne CPU developments. Dr. Cheng received an M.S. in Scientific (Biomedical Electronics) Instrumentation, UC Santa Barbara, 1975; and a Ph.D. in Computer Engineering, UC Santa Cruz, 1996. He holds eight issued U.S. Patents.