

Writing Efficient Code for ARM

Chris Shore
Customer Training Manager
ARM

ARM DEVELOPERS'
CONFERENCE '06

From the abstract

“... inside knowledge of how the compiler works ... simple techniques which will greatly improve ... object code”

Efficiency?

- What does “efficient” mean?
 - Code execution speed
 - Data throughput
 - Code or data size
 - Power consumption

- All are valid criteria against which we can measure success



3



Agenda

- General tools considerations

Platform issues

Efficient coding strategies



4



Tools issues

- First of all, some features of the tools which can help
- Compiler
 - Compiler optimizations
 - Multifile compilation
 - ARM extensions to C
 - Including assembler
- Linker
 - Feedback files
 - Linker optimizations
- Libraries
 - Real-time division



5



Code generation control

- General optimization
 - -O0 Minimum optimization – Leave my code alone
 - -O1 Restricted optimization – I need to debug this too
 - -O2 High optimization – Do what you like...
 - -O3 Maximum optimization – ...and then some!

 - -Ospace/-Otime
 - Obvious really, -Ospace is the default

- Specific features
 - --no_inline Disables inlining
 - --split_ldm Limits max registers in LDM/STM to five
 - --split_sections Puts each function in its own output section




6



Space and Time?

- Space/Time – what's the difference?
 - Space
 - Will attempt to reduce code size at the expense of execution speed
 - Will inline less often
 - Structure copying uses helper functions rather than inline code
 - Time
 - Will attempt to increase speed at the expense of code size

```
while (expr)
{
    body;
}
```



```
if (expr) do
{
    do
    {
        body;
    } while (expr);
}
```

ARM

7

ARM DEVELOPERS'
CONFERENCE '06

Level 2/Level 3?

- O2/O3 – what's the difference?
 - Inlines more aggressively
 - Multifile compilation enabled by default
 - Beware of effect on build time
 - High-level scalar optimizations e.g. loop unrolling
 - Enabled for “-O3 -Otime”
 - Code size cost is modest, build time cost can be large

ARM

8

ARM DEVELOPERS'
CONFERENCE '06

New loop optimizer

- “-O3 -Otime” enables the new loop optimizer
- This is either fully enabled or fully disabled
 - There is no documented finer level of control
- Automatically restructures loops
 - Loop unrolling, fusion, interchange, rotation and switching
 - Idiom recognition
 - Pointerization



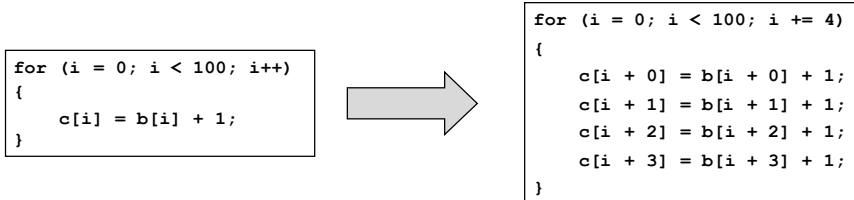
How to get the most out of it

- Use “restrict” pointer modifier as much as possible
 - Tells the compiler that there are no aliased accesses to addressed data (more on the details of this later...)
 - Can be applied to function arguments as well as local declarations
- Write simple loop bodies which access simple arrays in simple ways
 - i.e. with simple stride patterns
 - Rightmost subscript should be index of innermost loop in nested loops
- Don't try to do it manually!
 - In the worst case, this will confuse the optimization engine resulting in sub-optimal code generation
 - Best case, it will simply recognize (and possibly undo) your attempt



Loop unrolling

- Reduces loop overhead at direct cost of code size

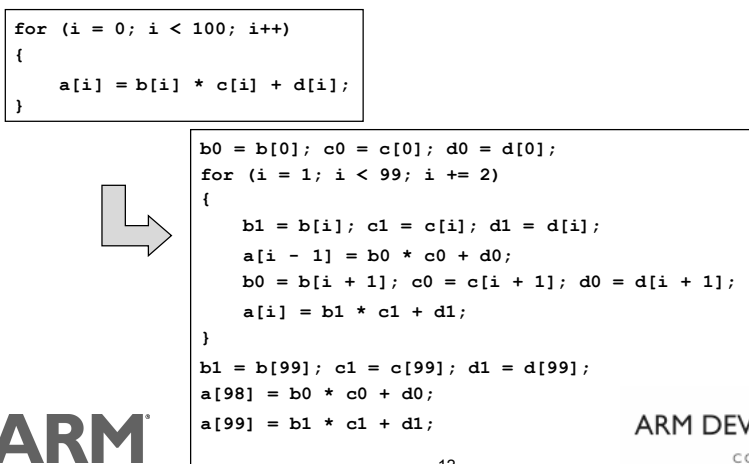


- Loop re-rolling
 - Recognise manually unrolled loops, re-roll and unroll optimally
- Complete unrolling
 - Recognise constant, low trip count loops and totally unroll removing all loop overhead



Loop rotation

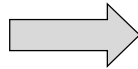
- Rotate unrolled loops to allow better instruction scheduling
 - Memory latency of one iteration overlaps with computation of another



Loop fusion

- Fuse small identical trip count loops together
 - Share common code
 - Reduce loop overhead

```
for (i = 0; i < 100; i++)  
{  
    c[i] = b[i] + 1;  
}  
for (j = 0; j < 100; j++)  
{  
    a[j] = b[j] - 1;  
}
```

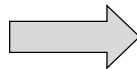


```
for (i = 0; i < 100; i++)  
{  
    c[i] = b[i] + 1;  
    a[i] = b[i] - 1;  
}
```

Loop Unswitching

- Remove invariant conditionals from loop bodies by replicating the loops and moving the tests outside

```
for (i = 0; i < 100; i++)  
{  
    if (operation == IADD)  
    {  
        c[i] = a[i] + b[i];  
    }  
    else  
    {  
        c[i] = a[i] * b[i];  
    }  
}
```



```
if (operation == IADD)  
{  
    for (i = 0; i < 100; i++)  
    {  
        c[i] = a[i] + b[i];  
    }  
}  
else  
{  
    for (i = 0; i < 100; i++)  
    {  
        c[i] = a[i] * b[i];  
    }  
}
```

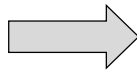
Pointerization

- Convert loops to increment pointers and count downwards rather than indexing into arrays and counting up
 - Eliminates address calculations
 - Simplifies loop termination conditions

```

for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];

start
    LDR r0, [r2, r3, LSL #2]
    LDR r1, [r4, r3, LSL #2]
    ADD r0, r0, r1
    STR r0, [r5, r3, LSL #2]
    ADD r3, r3, #1
    CMP r3, r3, #n
    BLT start
    
```



```

int *a1 = a;
int *b1 = b;
int *c1 = c;
for (i = n; i != 0; i--)
    a++ = b++ + c++;

start
    LDR r0, [r2], #4
    LDR r1, [r4], #4
    ADD r0, r0, r1
    STR r0, [r5], #4
    SUBS r3, r3, #1
    BNE start
    
```

Idiom Recognition

- Replace conditionals with min/max functions:

```
if (a > b) c = b; else c = a;    →    c = __min(a,b);
```

- Replace saturations with intrinsics:

```
if (a > 255) a = 255;           →    a = __sat(a,255);
```

- Replace loops with optimized library calls:

```
for (int i=0; i<256; i++) *a++=*b++; →    memcpy(a,b,256);
```

- Also memclr, memset & strcpy

Example

- This loop has been
 - Partially unrolled
 - Pointerized
 - Inverted
 - Rotated

```
void increment(int *restrict b,
              int *restrict c)
{
    int i;
    for (i = 0; i < 100; i++)
    {
        c[i] = b[i] + 1;
    }
}
```

Optimized version is 13 instructions compared to 8 for unoptimized

Takes 23% fewer instructions in 34% fewer cycles

```
void increment(int *b, int *c)
{
    int i;
    int *pb, *pc;
    int b3, b4;

    pb = b - 1;
    pc = c - 1;

    b3 = pb[1];

    for (i = (100 / 2); i != 0; i--)
    {
        b4 = *(pb += 2);
        pc[1] = b3 + 1;
        b3 = pb[1];
        *(pc += 2) = b4 + 1;
    }
}
```

ARM

17

ARM DEVELOPERS
CONFERENCE '06

Summary

- Points to remember
 - Enabled automatically with “-O3 -Otime”
 - Use *restrict* modifier as much as possible
 - Write plain and simple code
- Debugging
 - Combinations of these optimizations can result in very different program flow
 - Debugging should be done at low optimization levels first
 - User code must not rely on undefined behavior
- Metrics
 - Over EEMBC, improves performance by about 10%
 - Costs less than 1% in code size
 - 30% increase in compilation time

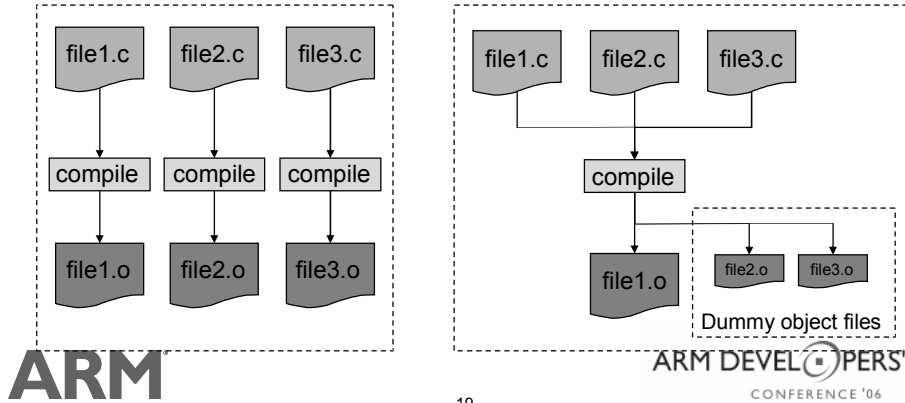
ARM

18

ARM DEVELOPERS
CONFERENCE '06

Multifile compilation

- Allows the compiler to optimize across source files
 - Increased inlining possibilities
 - Better base pointer and cross-function optimization
 - Reduced scatter file flexibility



19

ARM extensions

- `__restrict`
 - Supported as in C99 standard
- `__pure`
 - Non-standard extension
- `__value_in_regs`
 - Allows a function to pass back result in registers
 - Can speed up returning a structure up to 4 words in size

__restrict

- Allows you to tell the compiler that pointers do not reference overlapping areas of memory
- Available as “restrict” if enabled with --restrict

```
void copymem(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
```

- In this code segment a and b are guaranteed by the programmer to reference different regions of memory
- Beware of making promises which you can't (or won't) keep...!



21

__pure

- A function can be described as “pure” if:
 - Its result depends exclusively on its arguments
 - It has no side-effects
- A pure function may not, therefore:
 - Use global variables
 - Dereference pointers
 - Access memory except the stack
- Declaring a function as “pure” makes it a candidate for “common sub-expression elimination”

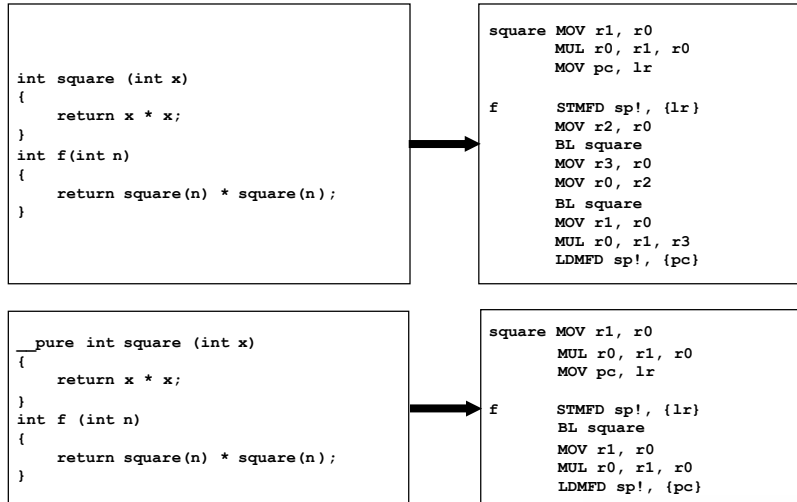
```
__pure int f(int arg)
```

- __pure is an ARM-specific extension to ANSI C



22

__pure (2)

**ARM**

23

ARM DEVELOPERS'
CONFERENCE '06

__value_in_regs

- Returns multi-word value in multiple registers

```

typedef struct int64_struct
{
    unsigned int lo;
    unsigned int hi;
} int64_struct;

__value_in_regs extern
int64_struct mul64(unsigned a, unsigned b);

```

- Avoids returning by copy on stack
- Useful for calling assembler functions which return more than one value
- Will be ignored (with warning) if return value is too large

ARM

24

ARM DEVELOPERS'
CONFERENCE '06

Linking to assembler

- C-Assembly linkage
 - Ensure compliance with AAPCS and all should be well...
- AAPCS (Procedure Call Standard for the ARM Architecture)
 - Part of the ABI (Application Binary Interface)
 - Specifies use of registers at procedure call interface

r0	Parameters in Values out (Corrupted)
r1	
r2	
r3	
r4	Internal variables (Preserved)
r5	
r6	
r7	
r8	
r9	Platform register
r10	Internal variables (Preserved)
r11	
r12 (ip)	Scratch register
r13 (sp)	
r14 (lr)	
r15 (pc)	

ARM

25

ARM DEVELOPERS' CONFERENCE '06

Inline assembler

- Generally used for short sequences which cannot be produced directly from C
- Beware...
 - It's not a "true" assembler
 - Some restrictions in functionality
 - The code is optimized along with the surrounding C
 - Register names are "virtualized"
- Gradually being replaced by intrinsics

```

__inline void enable_IRQ
{
    int tmp;
    __asm
    {
        MRS    tmp, CPSR
        BIC    tmp, tmp, #0x80
        MSR    CPSR_c, tmp
    }
}
    
```

Note that this is an example of a function which can now be performed using an intrinsic

ARM

26

ARM DEVELOPERS' CONFERENCE '06

Intrinsics

- Built-in ARM extensions to the C/C++ language
- Used to access machine features in a portable way
- Can accomplish many things which inline assembler can't
 - And in a much more portable, future-proof way

```
void disable_irq(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        ORR tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}
```



```
void __disable_irq(void);
```

Others include...

```
__enable_fiq()
__breakpoint()
__nop()
__current_sp()
__return_address()
__current_pc()
```

ARM

27

ARM DEVELOPERS'
CONFERENCE '06

New intrinsics

- `__semihost`
 - Architecture-independent way to generate semihosting calls (e.g. uses BKPT on v7M cores, SWI/SVC on everything else)
- `__schedule_barrier`
 - Creates a sequence point from the point of view of optimisation
 - Similar to `nop()` intrinsic except that no NOP instruction is generated
- `__force_stores`
 - Forces all globally-visible variables to be written to memory if they have been changed
- `__memory_changed`
 - Forces all globally-visible variables to be written back to memory if they have changed and then all read back from memory

ARM

28

ARM DEVELOPERS'
CONFERENCE '06

Support for the NEON instruction set

- The NEON instruction set is implemented by the Cortex-A8
- A set of vector data types e.g.
 - `int16x4_t` – a vector of 4 16-bit signed integers
 - `int32x2_t` – a vector of 2 32-bit signed integers
- A set of intrinsic functions e.g.
 - `vld1_s16` – load 4 16-bit signed values into a vector
 - `vadd_s16` – add two 4-element 16-bit vectors

```
int16x4_t vec;           /* declare vector data */
vec = vld1_s16(array);  /* load 4 values in parallel from the array */
array += 4;            /* increment the array pointer */
acc = vadd_s16(acc, vec); /* add the vector to the accumulator vector */
```



Embedded assembler

- Declare asm functions in C/C++ modules
 - With full function prototypes, including arguments and return value
 - Can access C preprocessor and structure offset information directly
 - Possible to insert Thumb assembler functions in an ARM module and vice versa
 - Processed by the “real” assembler
 - Supports full instruction set
 - C/C++ expressions can be embedded using `__cpp`
 - Useful for writing larger performance-critical functions
- Cannot be inlined

```
__asm void scpy(char *s, char *d)
{
loop
    LDRB r3, [r0], #1
    STRB r3, [r1], #1
    CMP r3, #0
    BNE loop
    BX lr
}
```



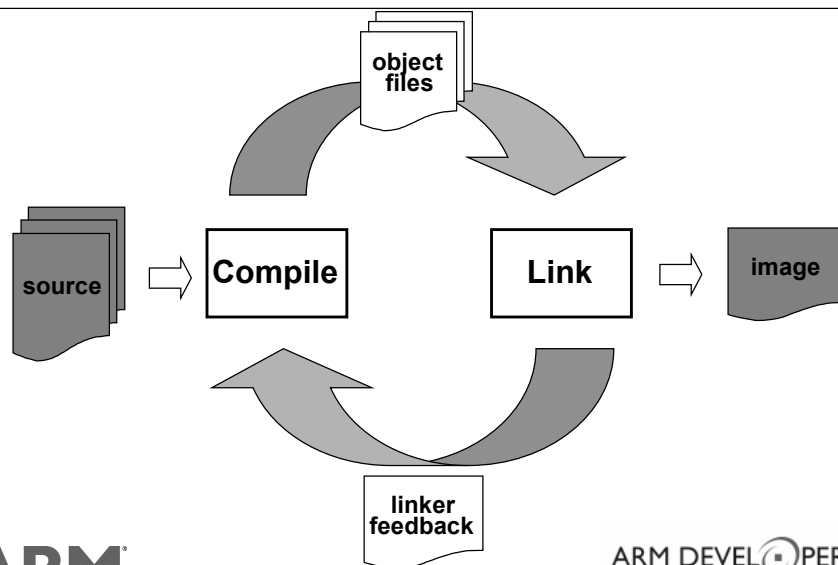
Linker optimizations

- The linker includes several features to improve output code
- Linker feedback files
 - Feedback file produced by linker and used as input to compilation
 - Largely replaces the `--split_sections` method and has better results
 - Removes unused code
 - Can reduce overhead of interworking
 - Removes functions which have been inlined everywhere they are called
- Small function inlining
 - Single line functions will be placed over the calling branch instruction
- Tail re-ordering
 - Modules will be reordered so that a tail-call can be replaced with a NOP if possible



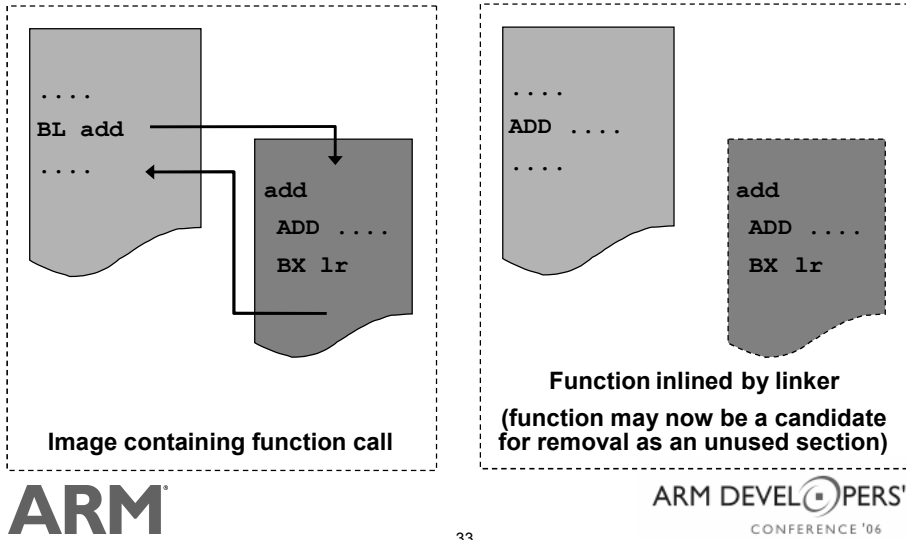
31

Linker feedback



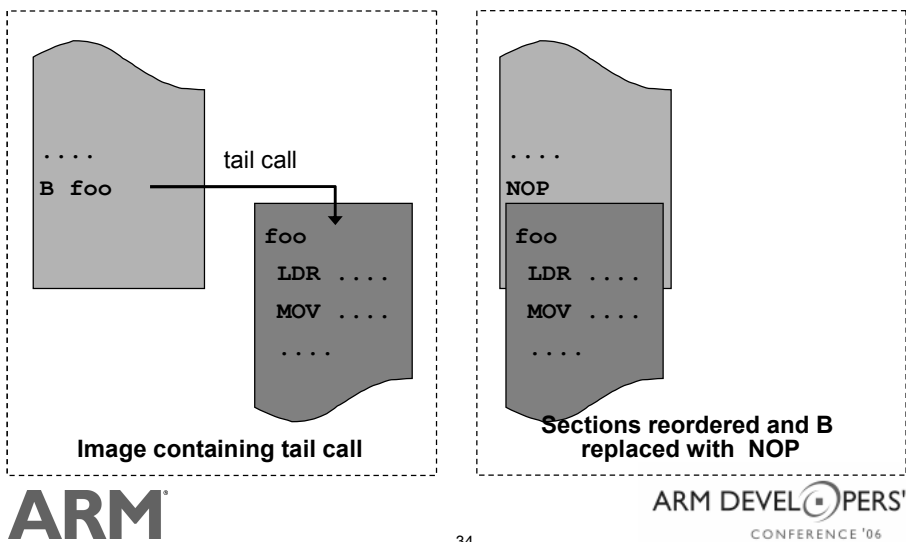
32

Small function inlining (--inline)



33

Tail re-ordering (--tailreorder)



34

Agenda

General tools considerations

- Platform issues

Efficient coding strategies



35



Starting from scratch

- Algorithm selection
 - What works best on ARM?
 - Making best use of the available register set and instruction set
- Data management
 - Mapping data to memory
 - Alignment issues
- Cache considerations
 - Cache improves everything except TCM
 - ARM10 and ARM11 cores work a lot better out of cache
- Writing exception handlers
- Efficient coding strategies
- Much of this will apply when working on existing code too...



36



Algorithm selection

- The ARM has a large (but not enormous) register set
- Memory access is (relatively) slow

- Algorithms which maintain as much as possible in registers will generally perform better
 - Unroll loops to make better use of registers
 - Avoid aliasing and use “restrict” where appropriate/possible
- All registers are 32-bit
 - Processing word-sized values is more efficient
 - If your data is sub-word, it is often possible to process more than one item at a time using SIMD techniques
 - Later ARM cores support many SIMD instructions



37

Data organization

- Some fast memory will help a lot

- Especially when programming in C, locating the stack in fast memory will speed up code considerably

- Avoid unaligned data wherever possible
 - ARM11 supports unaligned access in hardware
 - It's still slower than aligned access, though
 - And you still need to tell the compiler about it
 - Otherwise the compiler will carry out some “optimizations” which may break your code



38

Porting from one ARM to another

- What new features are available?
 - New instructions
 - Other new architectural features or behavior
- What else has changed?
 - Exception handling model
 - ARM7 cores implement “Base-Updated Abort Model”
 - Later cores automatically restore the base register
 - Caches
 - Tightly Coupled Memory
 - A “real-time” alternative to cache
 - Can be used to guarantee performance for critical areas without the uncertainty of cache




Make full use of the instruction set

- For example...
 - ...Architecture v6 contains a lot of new instructions
 - Packed data
 - Word/Halfword reversal
 - SIMD operations
 - More efficient exception entry/exit
 - Improved support for saturated maths
 - Support for mixed-endian systems
 - Load/Store exclusive for synchronization primitives
- Lots of new features too in Thumb-2, NEON, ARMv7...

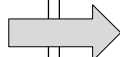


Example – Re-entrant interrupt

<pre> V5TE_Handler ; save lr and spsr SUB lr, lr, #4 STMFD sp!, {lr} MRS r14, SPSR STMFD sp!, {r12, r14} ; change to system mode with IRQ enabled MRS r14, CPSR BIC r14, r14, #0x9F ORR r14, r14, #0x1F MSR CPSR_c, r14 ; save user mode regs and call C handler STMFD sp!, {r0-r3, lr} BL C_irq_handler LDMFD sp!, {r0-r3, lr} ; change to IRQ mode with IRQ disabled MRS r12, CPSR BIC r12, r12, #0x1F ORR r12, r12, #0x92 MSR CPSR_c, r12 ; restore regs and return LDMFD sp!, {r12, r14} MSR SPSR_c, r14 LDMFD sp!, {PC}^ </pre>		<pre> V6_Handler ; save lr and spsr SUB lr, lr, #4 SRSEFD #SYSmode! ; change to system mode with IRQ enabled CPSIE i, #SYSmode ; save user mode regs and call C handler STMFD sp!, {r0-r3, r12, lr} BL C_irq_handler LDMFD sp!, {r0-r3, r12, lr} ; restore regs and return RFEED sp! </pre>
---	---	---



Example - Motion Estimation

<pre> ; This code processes four pixels loaded as ; two words, using V5TE instructions ; ; Excluding the loads, it takes (3 * 6) + 4 = 22 ; cycles per four pixels ; MOV Sum, #0 ; clear accumulator ; ; load four pixels LDR Rx, [Rxptr, #offset] LDR Ry, [Ryptr, #offset] ; process first pixel in pair of words MOV temp, Rx, LSR #24 ; get top byte SUBS temp, temp, Ry, LSR #24 ; difference RSBMI temp, temp, #0 ; abs difference ADD Sum, Sum, temp ; accumulate ; repeat following block three times to process ; remaining pixels MOV Rx, Rx, LSL #8 ; discard used pxl MOV Ry, Ry, LSL #8 ; discard used pxl MOV temp, Rx, LSL #24 ; get top byte SUBS temp, temp, Ry, LSR #24 ; difference RSBMI temp, temp, #0 ; abs difference ADD Sum, Sum, temp ; accumulate </pre>		<pre> ; This code processes four pixels loaded as ; two words, using the V6 SAD instruction ; ; Excluding the loads, it takes 1 cycle per ; four pixels ; MOV Sum, #0 ; ; load four pixels LDR Rx, [Rxptr, #offset] LDR Ry, [Ryptr, #offset] ; ; calculate and accumulate SAD for all four ; pixels USADA8 Sum, Rx, Ry, Sum </pre>
---	---	---



Porting to a Thumb-2 core

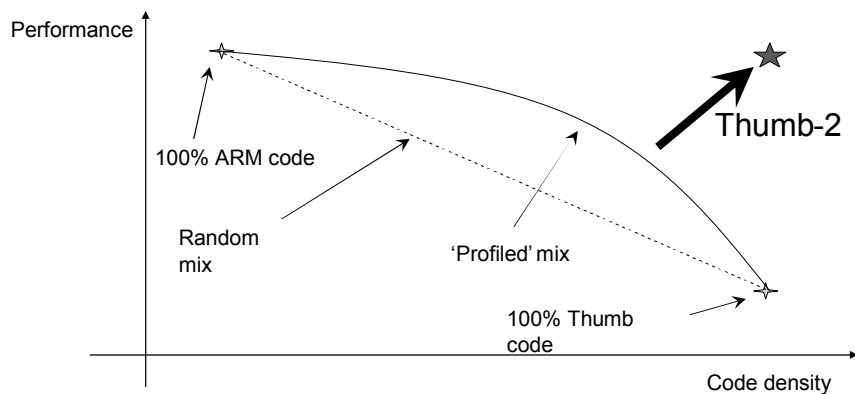
- Thumb-2 is a blend of 16- and 32-bit instructions
- No need to manually select between ARM and Thumb
- All code can be compiled in Thumb to get best mix
- Intrinsic in C compiler mean inline/embedded assembler can be avoided almost completely
 - Removes many portability issues
 - Allows compiler to pick best state/size for instructions
- Assembler should be written in UAL for best results
 - Allows assembler to select best output instructions
 - Easily convertible to ARM-only

ARM

43

ARM DEVELOPERS'
CONFERENCE '06

Performance / Density



ARM

44

ARM DEVELOPERS'
CONFERENCE '06

Thumb-2 can be faster than Thumb

C Code:	Thumb Code:	Thumb-2 code:
<pre>typedef struct { int x, y, z; int a[1024], b[1024]; } S; void f(S *s) { int i; for (i = 0; i < 1024; i++) { s->a[i] = -1; s->b[i] = i; } }</pre>	<pre>PUSH {r4, r5} MOVS r1, #0 MOVS r3, #1 LSLS r3, r3, #10 LSLS r4, r3, #2 SUBS r5, r1, #1 loop LSLS r2, r1, #2 ADDS r2, r2, r0 STR r5, [r2, #0xc] ADDS r2, r2, r4 STR r1, [r2, #0xc] ADDS r1, r1, #1 CMP r1, r3 BLT loop POP {r4, r5} BX lr</pre>	<pre>MOVS r1, #0 SUBS r3, r1, #1 loop ADD r2, r0, r1, LSL #2 STR r3, [r2, #0xc] ADD r2, r2, #0x1000 STR r1, [r2, #0xc] ADDS r1, r1, #1 CMP r1, #0x400 BLT loop BX lr</pre>
	<i>Total: 16 instns, 8 in loop</i>	<i>Total: 10 instns, 7 in loop</i>



Porting to Cortex-M3

- Cortex-M3 is a special case since there are several fundamental differences with other ARM cores
 - Exception model
 - PSRs
 - Thumb-2 only
 - UAL strongly recommended
 - See next slide
 - 99.5% can be written in C
 - Exception handlers
 - Startup code



Unified Assembler Language

- UAL gives the ability to write assembler code for all ARM processors that can have the execution state decided at assembly time
 - Previously code had to be written exclusively for ARM or Thumb state
 - Legacy assembler code will still assemble successfully
- The UAL defines effective 'pseudo' instructions that are resolved by the Assembler
 - The assembler will generate the machine code dependent upon the inline directives (e.g. `THUMB`) or the assembler switches (e.g. `--arm`)
- General rules for UAL
 - Use of `POP`, `PUSH`
 - Relaxation of register definitions for `Rd` and `Rs`
- See complete definition in RVCT Assembler Guide



UAL Changes

Traditional	UAL
<pre> Asub STMFD sp!, {r4, r5, lr} ADDEQS r0, r0, r3 MOV r1, r2 LSL #4 BL Bsub LDMFD sp!, {r4, r5, pc} Bsub STMIA r0, {r2, r3} LDMIA r1, {r2, r3} LDR r4, [pc,#0x20] BX lr Value DCD 0x8000 </pre>	<pre> Asub PUSH {r4, r5, lr} ADDSEQ r0, r3 LSL r1, r2, #4 BL Bsub POP {r4, r5, pc} Bsub STM r0, {r2, r3} LDM r1, {r2, r3} LDR r4, Value BX lr Value DCD 0x8000 </pre>



UAL use in ARM Tools

- ARM RVDS 2.2+ compilation, link and debug tools have been developed to use the UAL definition
- Assemblers will accept BOTH the old and new syntax
- The fromelf utility will output a subset of the complete UAL
 - PUSH and POP are shown in place of the STMFD/LDMFD instructions
 - Any MOV that only uses the shift operation is shown as the shift in the output

MOV r0, r0, LSL #4 is shown as LSL r0, #4

- It does not automatically relax registers in data processing instructions where the source and destination registers are the same
- The SWI instruction decodes as SVC



49



Agenda

General tools considerations

Platform issues

- Efficient coding strategies



50



Efficient coding strategies

- Knowledge of the machine and the compiler can help write efficient code in several areas
- By writing your code carefully, you can give the compiler the best shot at producing efficient output

- Beware, though, of using the compiler as a “high-level assembler!”

- The following is general advice
 - Supplementary information is included after the end of this presentation for your reference



Loops Should Count Down

- **Loops which count down to zero are more efficient**
 - Compare with zero is usually free in ARM instruction set
 - The limit value is only needed at the start and need not occupy a register

<pre> unsigned int fact1(unsigned int limit) { unsigned int i; unsigned int fact = 1; for (i = 1; i <= limit; i++) { fact = fact * i; } return fact; } fact1 0x000000 : MOV r2,#1 0x000004 : MOV r1,#1 0x000008 : CMP r0,#1 0x00000c : BGT 0x20 0x000010 : MUL r2,r1,r2 0x000014 : ADD r1,r1,#1 0x000018 : CMP r1,r0 0x00001c : BLE 0x10 0x000020 : MOV r0,r2 0x000024 : MOV pc,lr </pre>		<pre> unsigned int fact2(unsigned int limit) { unsigned int i; unsigned int fact = 1; for (i = limit; i != 0; i--) { fact = fact * i; } return fact; } fact2 0x000000 : MOVS r1,r0 0x000004 : MOV r0,#1 0x000008 : MOV pc,lr 0x00000c : MUL r0,r1,r0 0x000010 : SUBS r1,r1,#1 0x000014 : BNE 0x0c 0x000018 : MOV pc,lr </pre>
--	--	---

This code is compiled with “-O2 -Otime”



Loop Counters

- Unsigned int for loop counter if possible
- Compare for equality with zero rather than > 0

- Use

```
unsigned int j;           //          MOV    j, #10
for (j = 10; j != 0; j--) // for_loop ...
                                SUBS   j, j, #1
                                BNE    for_loop
```

- Not

```
int j;                   //          MOV    j, #10
for (j = 10; j > 0; j--) // for_loop ...
                                SUB    j, j, #1
                                CMP    j, #0
                                BGT    for_loop
```

ARM

53

ARM DEVELOPERS'
CONFERENCE '06

Parameter Passing

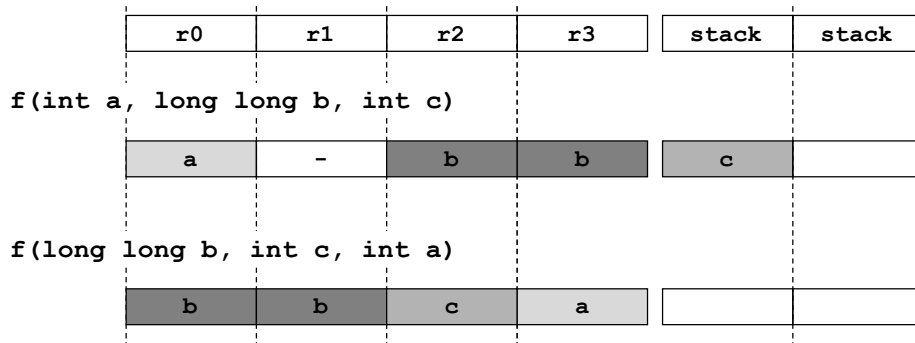
- Keep parameters to four or fewer
- “ARM/Thumb Procedure Call Standard”
 - Allows up to four integer-sized parameters to be passed in registers
 - Further parameters will be passed on the stack
- Stack accesses are costly in terms of space and time
- Note that C++ passes the “this” pointer as a hidden parameter so only three registers remain available for other parameters

ARM

54

ARM DEVELOPERS'
CONFERENCE '06

Register usage in parameter lists



General advice

- Avoid division
 - ARM cores (except some Cortex cores) have no division hardware
 - Remember that modulo is effectively a division operation

- Access to unaligned data is dangerous and/or expensive
 - For efficient use, data items should be aligned on natural boundaries
 - v6 architecture introduces unaligned support in hardware but software engineers still need to be careful...

- Registers are 32-bit
 - Sub-word quantities may not be handled efficiently



Alignment of pointers

- Be **VERY** careful with alignment of pointers
 - Can lead to runtime failures

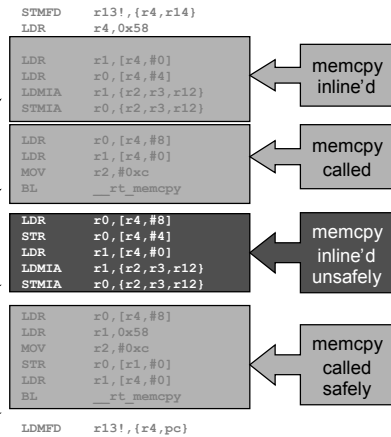
```
#include <string.h>
int *a = (int *)0x1000;
int *b = (int *)0x2000;
char *c = (char *)0x3001;
__packed int *d;

void foo (void)
{
    memcpy (b,a,12);

    memcpy (c,a,12);

    b = (int *)c;
    memcpy (b,a,12);

    d = (__packed int *)c;
    memcpy ((void *)d,a,12);
}
```



Consider Element Offsets

- To calculate the address of an element of an array, the compiler must multiply the size of the element by the index...

$$\&(a[i]) = a + i * \text{sizeof}(a)$$

- If the element size is a power of 2, this can be done with a simple inline shift

- For an array at [r3], to access the word at index r1

- Element size = 12:

```
ADD r1, r1, r1, LSL #1 ; r1 = 3 * r1
LDR r0, [r3, r1, LSL #2] ; r0 = *(r1 + 4 * r1)
```

- Element size = 16:

```
LDR r0, [r3, r1, LSL #4] ; r0 = *(r3 + 16 * r1)
```



Local Variables

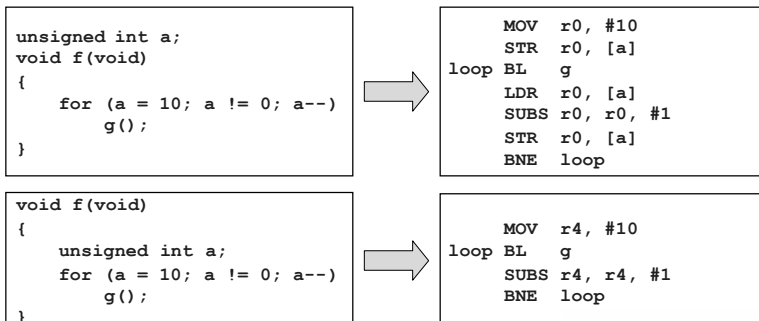
- Local variables are held in registers wherever possible...
 - ...but in some circumstances they must be placed on the stack
- This is much less efficient

- When can variables be held in registers?
 - When there are not too many of them
 - When their address is not known to the program
- So...
 - Minimize the number of “live” local variables
 - Avoid taking the address of a local variable
- This is more important in Thumb state as there are fewer registers available to the compiler

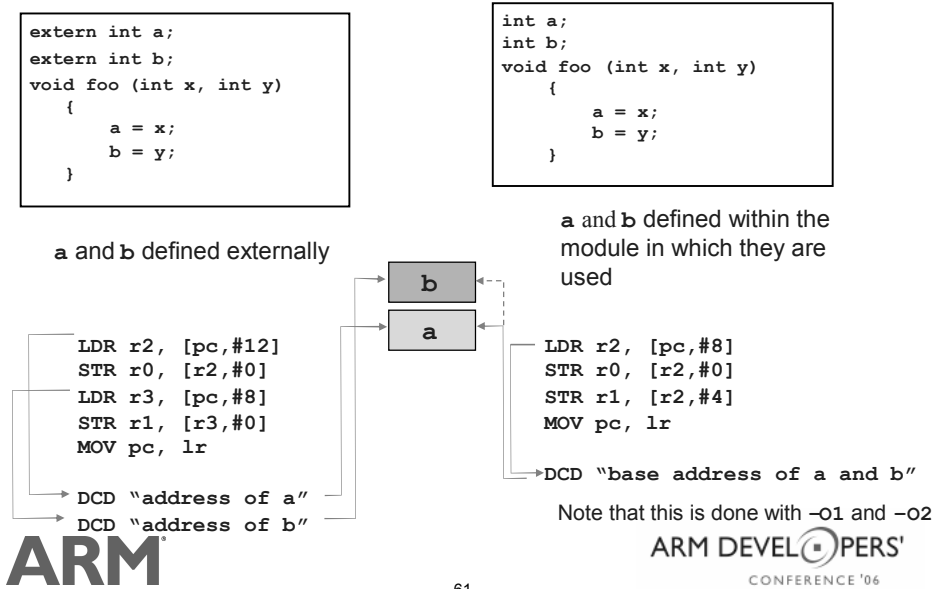


Global Data

- Global (non-automatic) data is declared at module level
- It is allocated static storage in RAM
 - May also occupy ROM space if initialized
- Global data use is much less efficient compared to local data use
- Here is an example using a global variable as a loop counter



Base Pointers



61

Enabling Base Pointer Optimization

- If globals are placed into a structure, then access to each element of structure will naturally be as an offset from a single base pointer.
 - Elements in struct will be aligned on size boundaries
 - The compiler will not re-order the structure
- Group data into several 'logical' structures rather than one large structure.
- '#define' can be used to allow change to be hidden from main application code.
 - #define value mystruct.value

External Globals

data.c

```
int a;
int b;
```

code.c

```
extern int a;
extern int b;

int main(void)
{
    return a + b;
}
```

Assembler output

```
main      LDR      r0,0x000080c0
000080ac  LDR      r1,0x000080c4
000080b0  LDR      r0,[r0,#0]
000080b4  LDR      r1,[r1,#0]
000080b8  ADD      r0,r0,r1
000080bc  MOV      pc,lr
000080c0  DCD      0x000083d4
000080c4  DCD      0x000083d8
```

```
struct data
{
    int a;
    int b;
} mystruct;
```

```
extern struct data mystruct;

int main(void)
{
    return mystruct.a + mystruct.b;
}
```

```
main      LDR      r0,0x000080bc
000080ac  LDR      r1,[r0,#0]
000080b0  LDR      r0,[r0,#4]
000080b4  ADD      r0,r1,r0
000080b8  MOV      pc,lr
000080bc  DCD      0x000083cc
```



Further Information

- ARM Related Books
 - www.arm.com
 - Technical Support → Documentation → Books
- ARM Application Notes
 - Technical Support → Documentation → Application Notes
- ARM Training
 - Technical Support → Training
 - training@arm.com
- Ask me!



Writing Efficient Code for ARM

Chris Shore
Customer Training Manager
ARM

ARM DEVELOPERS'
CONFERENCE '06

Tail-call Optimization

- **Procedure calls are potential sources of inefficiency**
 - Branch and return use three cycles each
 - Often involve stack access
 - Parameters may be placed on stack
 - Local variables of calling and called routines are on stack
- **Eliminating calls and/or returns saves time and stack space**
 - Tail-call optimization allows a procedure call to be replaced with a “goto” when it is the last statement of a function
 - Avoids unnecessary return (saves time)
 - Reduces stack usage
 - Caller stack frame can be deleted prior to call
 - Reduced stack access saves time

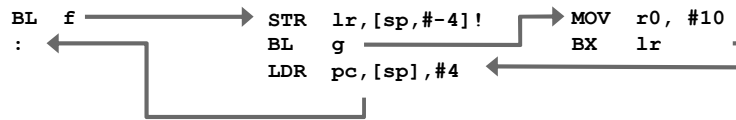
Tail-call Optimization Example

- Tail call optimization avoids the use of unnecessary returns in function hierarchies
 - BL translated to B where possible
 - Enabled at high optimization (-O1, -O2)

```
int main()
{
    int x = f();
    :
}

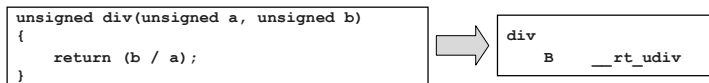
int f()
{
    int y = g();
    return y;
}

int g()
{
    return 10;
}
```

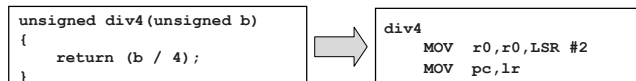


Division Operations (1)

- The ARM core contains no division hardware
 - Division will typically be implemented by a run-time library function.
 - Many cycles to execute



- Typical execution time for this is 12-96 cycles, average c.30
- The `__use_realtime_division` option will use a more predictable division function (always < 45 cycles but slower for typical values)
- Division by compile-time constants are treated as a special case
 - Division by powers of two will use shift operations

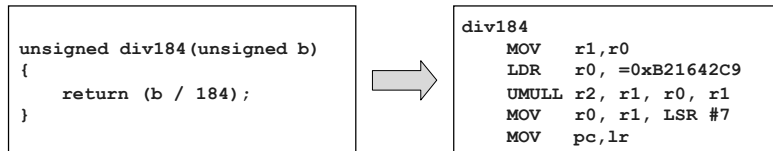


ARMv7M and ARMv7R cores do include division in hardware



Division Operations (2)

- Division by other constants
 - The compiler will use multiplication to implement division by constants



- 0xB21642C9 is a fixed-point binary representation of 1/184
- UMULL is used to multiply the argument by 1/184
- LSR #7 is used to normalize the result

Modulo Operations

- The remainder operator '%' is commonly used in modulo arithmetic
- However, this will be expensive if the modulo value is not a power of two
 - Avoid by rewriting C code to use `if()` statement check
- For example, if count has the range 0 to 59, replace:

`count = (count+1) % 60;` with `if (++count >= 60) count = 0;`

<pre>modulo ADD r1,r0,#1 MOV r0,#0x3c BL __rt_udiv MOV r0,r1</pre>

<pre>test_and_reset ADD r0,r0,#1 CMP r0,#0x3c MOVCS r0,#0</pre>
--

This code is compiled with "-O1 -Ospace"

Unaligned Data Access

- ARM LDR/STR instructions can access data aligned on natural boundaries
- Access to unaligned data can be done but is expensive

```
; unaligned read
LDRB r1, [r0, #0]
LDRB r2, [r0, #1]
LDRB r3, [r0, #2]
LDRB r4, [r0, #3]
ORR  r1, r1, r2, LSL #8
ORR  r1, r1, r3, LSL #16
ORR  r0, r1, r4, LSL #24
```

```
; unaligned write
STRB r0, [r1, #0]
MOV  r2, r0, LSR #8
STRB r2, [r1, #1]
MOV  r2, r0, LSR #16
STRB r2, [r1, #2]
MOV  r2, r0, LSR #24
STRB r2, [r1, #3]
```

- Used `__packed` qualifier to indicate (possible) non-alignment to compiler

v6 cores include hardware support for unaligned memory access

ARM

ARM DEVELOPERS'
CONFERENCE '06

71

Unaligned support in ARMv6

- Unaligned access only supported for LDR/STR
 - Other accesses must maintain alignment (LDRH, LDREX etc.)
- Not enabled by default
 - Though the tools will assume that it is
 - Use `-no_unaligned_access` to disable in RVCT
- Accesses are not guaranteed atomic at bus level
 - May cross cache/page/line boundaries
 - May be interrupted by another bus master
 - Therefore unsafe in Peripheral or Shared memory
- Must still use `__packed` to identify all potentially unaligned accesses in source code
 - Otherwise compiler will attempt to use LDM/LDRD

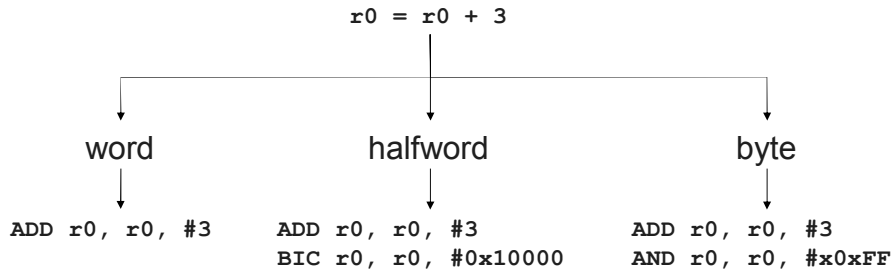
ARM

ARM DEVELOPERS'
CONFERENCE '06

72

Words Are Efficient

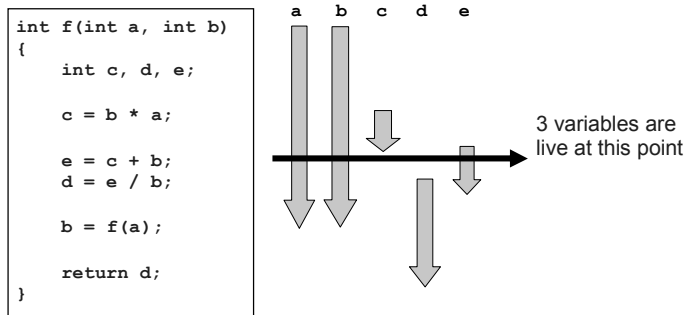
- Arithmetic
 - ARM registers are 32-bits
 - This makes 32-bit operations naturally efficient
 - 8-bit and 16-bit operations are generally less efficient



- The mask operations are necessary to remove potential overflow bits

“Live” Variables


- A “live” variable is one whose value must be maintained at a particular point in a procedure



- The values of up to 3 variables will have to be maintained at any one time in this example
- Note that the compiler can use same register for c, d and e

“Address-taken” Local Variables

- Taking the address of a local variable may force the compiler to keep it updated in memory (on the stack) rather than holding it in a register
- This can be avoided by using a temporary variable...

<pre>int f(int i) { g(&i); // use i return i; }</pre>		<pre>int f(int i) { int dummy = i; g(&dummy); i = dummy; // use i return i; }</pre>
---	---	--

- The second version of the function can hold i in a register throughout