

MICROCONTROLLER

ARM PROGRAMMING USING ASSEMBLY LANGUAGE

WRITING ASSEMBLY CODE:

This section gives examples showing how to write basic assembly code. Also, this section uses the *ARM macro assembler armasm* for examples.

Example 1:

<p>This example shows how to convert a <i>C</i> function to an assembly function—usually the first stage of assembly optimization. Consider the simple <i>C</i> program <i>main.c</i> following that prints the squares of the integers from 0 to 9:</p>	<p>Let's see how to replace <i>square</i> by an assembly function that performs the same action. Remove the <i>C</i> definition of <i>square</i>, but not the declaration (the second line) to produce a new <i>C</i> file <i>main1.c</i>. Next add an <i>armasm</i> assembler file <i>square.s</i> with the following contents:</p>
<pre>#include <stdio.h> int square(int i); int main(void) { int i; for (i=0; i<10; i++) { printf("Square of %d is %d\n", i, square(i)); } } int square(int i) { return i*i; }</pre>	<pre>AREA .text , CODE, READONLY EXPORT square ; int square(int i) square MUL r1, r0, r0 ; r1 = r0 * r0 MOV r0, r1 ; r0 = r1 MOV pc, lr ; return r0 END</pre>

- The *AREA* directive names the area or code section that the code lives in. If you use non-alphanumeric characters in a symbol or area name, then enclose the name in vertical bars. Many non-alphanumeric characters have special meanings otherwise. In the previous code we define a read-only code area called *.text*.
- The *EXPORT* directive makes the symbol *square* available for external linking. At line six we define the symbol *square* as a code label. Note that *armasm* treats non-indented text as a label definition.
- When *square* is called, the parameter passing is defined by the *ARM-Thumb procedure call standard (ATPCS)*. The input argument is passed in register *r0*, and the return value is returned in register *r0*. The multiply instruction has a restriction that the destination register must not be the same as the first argument register. Therefore we place the multiply result into *r1* and move this to *r0*.
- The *END* directive marks the end of the assembly file. Comments follow a semicolon.

The following script illustrates how to build this example using command line tools.

```
armcc -c main1.c
armasm square.s
armlink -o main1.axf main1.o square.o
```

Example 1 only works if you are compiling your *C* as ARM code. If you compile your *C* as Thumb code, then the assembly routine must return using a *BX* instruction.

Example 2: When calling ARM code from *C* compiled as Thumb, the only change required to the assembly in *Example 1* is to change the return instruction to a *BX*. *BX* will return to ARM or Thumb state according to bit 0 of *lr*. Therefore this routine can be called from ARM or Thumb. Use *BX lr* instead of *MOV pc, lr* whenever your processor supports *BX* (ARMv4T and above). Create a new assembly file *square2.s* as follows:

```
AREA    |.text|, CODE, READONLY

EXPORT  square

; int square(int i)
square
    MUL    r1, r0, r0    ; r1 = r0 * r0
    MOV    r0, r1        ; r0 = r1
    BX     lr            ; return r0
END
```

With this example we build the *C* file using the Thumb *C* compiler *tcc*. We assemble the assembly file with the interworking flag enabled so that the linker will allow the Thumb *C* code to call the ARM assembly code. You can use the following commands to build this example:

```
tcc -c main1.c
armasm -apcs /interwork square2.s
armlink -o main2.axf main1.o square2.o
```

Example 3: This example shows how to call a subroutine from an assembly routine. We will take Example 1 and convert the whole program (including main) into assembly. We will call the *C* library routine *printf* as a subroutine. Create a new assembly file *main3.s* with the following contents:

```

AREA    |.text|, CODE, READONLY

EXPORT  main

IMPORT  |Lib$$Request$$armlib|, WEAK
IMPORT  __main    ; C library entry
IMPORT  printf    ; prints to stdout

i      RN 4

        ; int main(void)

main
        STMFD    sp!, {i, lr}
        MOV      i, #0

loop
        ADR      r0, print_string
        MOV      r1, i
        MUL      r2, i, i
        BL       printf
        ADD      i, i, #1
        CMP      i, #10
        BLT      loop
        LDMFD    sp!, {i, pc}

print_string
        DCB      "Square of %d is %d\n", 0

END

```

- The *IMPORT* directive is used to declare symbols that are defined in other files.
- The imported symbol *Lib\$\$Request\$\$armlib* makes a request that the linker links with the standard ARM C library.
 - The *WEAK* specifier prevents the linker from giving an error if the symbol is not found at link time. If the symbol is not found, it will take the value zero.
- The second imported symbol *__main* is the start of the C library initialization code.

You only need to import these symbols if you are defining your own main; a main defined in C code will import these automatically for you. Importing *printf* allows us to call that C library function.

- The *RN* directive allows us to use names for registers. In this case we define *i* as an alternate name for register *r4*.

- Using register names makes the code more readable. It is also easier to change the allocation of variables to registers at a later date. Recall that *ATPCS* states that a function must preserve registers *r4* to *r11* and *sp*. We corrupt *i* (*r4*), and calling *printf* will corrupt *lr*. Therefore we stack these two registers at the start of the function using an *STMFD* instruction. The *LDMFD* instruction pulls these registers from the stack and returns by writing the return address to *pc*.

- The *DCB* directive defines byte data described as a string or a comma-separated list of bytes.

To build this example you can use the following command line script:

```
armasm main3.s
armlink -o main3.axf main3.o
```

Note that Example 3 also assumes that the code is called from ARM code. If the code can be called from Thumb code as in Example 2 then we must be capable of returning to Thumb code. For architectures before *ARMv5* we must use a *BX* to return. Change the last instruction to the two instructions:

```
LDMFD    sp!, {i, lr}
BX       lr
```

Example 4: This example defines a function *sumof* that can sum any number of integers. The arguments are the number of integers to sum followed by a list of the integers. The *sumof* function is written in assembly and can accept any number of arguments. Put the C part of the example in a file *main4.c*:

```
#include <stdio.h>

/* N is the number of values to sum in list ... */
int sumof(int N, ...);

int main(void)
{
    printf("Empty sum=%d\n", sumof(0));
    printf("1=%d\n", sumof(1,1));
    printf("1+2=%d\n", sumof(2,1,2));
    printf("1+2+3=%d\n", sumof(3,1,2,3));
    printf("1+2+3+4=%d\n", sumof(4,1,2,3,4));
    printf("1+2+3+4+5=%d\n", sumof(5,1,2,3,4,5));
    printf("1+2+3+4+5+6=%d\n", sumof(6,1,2,3,4,5,6));
}
```

Next define the *sumof* function in an assembly file *sumof.s*:

MICROCONTROLLER

```
AREA    |.text|, CODE, READONLY

EXPORT  sumof

N       RN 0    ; number of elements to sum
sum     RN 1    ; current sum

; int sumof(int N, ...)
sumof
    SUBS    N, N, #1        ; do we have one element
    MOVLT   sum, #0         ; no elements to sum!
    SUBS    N, N, #1        ; do we have two elements
    ADDGE   sum, sum, r2
    SUBS    N, N, #1        ; do we have three elements
    ADDGE   sum, sum, r3
    MOV     r2, sp          ; top of stack
loop
    SUBS    N, N, #1        ; do we have another element
    LDMGEFD r2!, {r3}       ; load from the stack

    ADDGE   sum, sum, r3
    BGE     loop
    MOV     r0, sum
    MOV     pc, lr          ; return r0

END
```

The code keeps count of the number of remaining values to *sum*, *N*. The first three values are in registers *r1*, *r2*, *r3*. The remaining values are on the stack (Recall that *ATPCS* places the first four arguments in registers *r0* to *r3*. Subsequent arguments are placed on the stack). You can build this example using the commands –

```
armcc -c main4.c
armasm sumof.s
armlink -o main4.axf main4.o sumof.o
```

PROFILING AND CYCLE COUNTING:

- ✓ The first stage of any optimization process is to identify the critical routines and measure their current performance. A ***profiler*** is a tool that measures the proportion of time or processing cycles spent in each subroutine. You use a profiler to identify the most critical routines.

- ✓ A **cycle counter** measures the number of cycles taken by a specific routine. You can measure your success by using a cycle counter to benchmark a given subroutine before and after an optimization.
- ✓ The ARM simulator used by the *ADSI.1 debugger* is called the *ARMulator* and provides profiling and cycle counting features.
 - The *ARMulator profiler* works by sampling the program counter *pc* at regular intervals. The profiler identifies the function the *pc* points to and updates a hit counter for each function it encounters. Another approach is to use the trace output of a simulator as a source for analysis.
 - The accuracy of a *pc*-sampled profiler is limited, as it can produce meaningless results if it records too few samples.
- ✓ ARM implementations do not normally contain cycle-counting hardware; so to easily measure cycle counts you should use an ARM debugger with ARM simulator.
 - You can configure the *ARMulator* to simulate a range of different ARM cores and obtain cycle count benchmarks for a number of platforms.

INSTRUCTION SCHEDULING:

The time taken to execute instructions depends on the implementation pipeline. For this section, we assume *ARM9TDMI* pipeline timings. The following rules summarize the cycle timings for common instruction classes on the *ARM9TDMI*.

Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

- ✓ *ALU* operations such as addition, subtraction, and logical operations take one cycle.
- ✓ This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the *pc*, then add two cycles.
- Load instructions that load *N* 32-bit words of memory such as *LDR* and *LDM* take *N* cycles to issue, but the result of the last word loaded is not available on the following cycle.
 - The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system. An *LDM* of a single value is exceptional, taking two cycles. If the instruction loads *pc*, then add two cycles.
 - Load instructions that load 16-bit or 8-bit data such as *LDRB*, *LDRSB*, *LDRH*, and *LDRSH* take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system.
- Branch instructions take three cycles.

- Store instructions that store N values take N cycles. This assumes zero-wait-state memory for an un-cached system, or a cache hit or a write buffer with N free entries for a cached system. An *STM* of a single value is exceptional, taking two cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product.

To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies. The *ARM9TDMI* processor performs five operations in parallel:

- **Fetch:** Fetch from memory the instruction at address pc . The instruction is loaded into the core and then processes down the core pipeline.
- **Decode:** Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.
- **ALU:** Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address $pc - 8$ (ARM state) or $pc - 4$ (Thumb state).
 - Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation.
 - Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.
- **LS1:** Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- **LS2:** Extract and zero- or sign-extend the data loaded by a byte or half-word load instruction. If the instruction is not a load of an 8-bit byte or 16-bit half-word item, then this stage has no effect.

The following Figure shows a simplified functional view of the five-stage *ARM9TDMI* pipeline.

Instruction address	pc	$pc-4$	$pc-8$	$pc-12$	$pc-16$
Action	Fetch	Decode	ALU	LS1	LS2

Note that multiply and register shift operations are not shown in the figure.

After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. Note that pc points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address $pc - 8$ in parallel with fetching the instruction at address pc .

How does the pipeline affect the timing of instructions? Consider the following examples. These examples show how the cycle timings change because an earlier instruction must complete a stage before the current instruction can progress down the pipeline.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a *pipeline hazard* or *pipeline interlock*.

Example 5: This example shows the case where there is no interlock.

```
ADD  r0, r0, r1
ADD  r0, r0, r2
```

This instruction pair takes two cycles. The ALU calculates $r0 + r1$ in one cycle. Therefore this result is available for the ALU to calculate $r0 + r2$ in the second cycle.

Example 6: This example shows a one-cycle interlock caused by load use.

```
LDR  r1, [r2, #4]
ADD  r0, r0, r1
```

This instruction pair takes three cycles. The ALU calculates the address $r2 + 4$ in the first cycle while decoding the *ADD* instruction in parallel. However, the *ADD* cannot proceed on the second cycle because the load instruction has not yet loaded the value of *r1*. Therefore the pipeline stalls for one cycle while the load instruction completes the *LS1* stage. Now that *r1* is ready, the processor executes the *ADD* in the ALU on the third cycle.

The following Figure illustrates how this interlock affects the pipeline.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	<i>ADD</i>	LDR	...
Cycle 3		...	ADD	—	LDR

The processor stalls the *ADD* instruction for one cycle in the ALU stage of the pipeline while the load instruction completes the *LS1* stage. Figure denotes this stall by *italic ADD*. Since the *LDR* instruction proceeds down the pipeline, but the *ADD* instruction is stalled, a gap opens up between them. This gap is sometimes called a pipeline *bubble*. We've marked the bubble with a *dash*.

Example 7: This example shows a one-cycle interlock caused by delayed load use.

```
LDRB  r1, [r2, #1]
ADD    r0, r0, r2
EOR    r0, r0, r1
```

This instruction triplet takes four cycles. Although the *ADD* proceeds on the cycle following the load byte, the *EOR* instruction cannot start on the third cycle. The *r1* value is not ready until the load instruction completes the *LS2* stage of the pipeline. The processor stalls the *EOR* instruction for one cycle. Note that the *ADD* instruction does not affect the timing at all. The sequence takes four cycles whether it is there or not! The following Figure shows how this sequence progresses through the processor pipeline. The *ADD* doesn't cause any stalls since the *ADD* does not use *r1*, the result of the load.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB	...	
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	EOR	ADD	LDRB
Cycle 4		...	EOR	—	ADD

Example 8: This example shows why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.

```

MOV    r1, #1
B      case1
AND    r0, r0, r1
EOR    r2, r2, r3
...
case1
SUB    r0, r0, r1

```

The three executed instructions take a total of five cycles. The *MOV* instruction executes on the first cycle. On the second cycle, the branch instruction calculates the destination address. This causes the core to flush the pipeline and refill it using this new *pc* value. The refill takes two cycles. Finally, the *SUB* instruction executes normally. The following Figure illustrates the pipeline state on each cycle. The pipeline drops the two instructions following the branch when the branch takes place.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Scheduling of Load Instructions:

Load instructions occur frequently in compiled code, accounting for approximately one-third of all instructions. Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance. The compiler attempts to schedule the code as best it can, but the aliasing problem of *C* limits the available optimizations. The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.

Consider an example of a memory-intensive task. The following function, *str_tolower*, copies a zero-terminated string of characters from *in* to *out*. It converts the string to lowercase in the process.

```
void str_tolower(char *out, char *in)
{
    unsigned int c;

    do
    {
        c = *(in++);
        if (c >= 'A' && c <= 'Z')
        {
            c = c + ('a' - 'A');
        }
        *(out++) = (char)c;
    } while (c);
}
```

```
str_tolower
    LDRB    r2,[r1],#1    ; c = *(in++)
    SUB     r3,r2,#0x41    ; r3 = c - 'A'
    CMP     r3,#0x19      ; if (c <= 'Z' - 'A')
    ADDLS   r2,r2,#0x20    ; c += 'a' - 'A'
    STRB    r2,[r0],#1    ; *(out++) = (char)c
    CMP     r2,#0         ; if (c!=0)
    BNE     str_tolower    ; goto str_tolower
    MOV     pc,r14        ; return
```

The compiler generates the above compiled output. Notice that the compiler optimizes the condition (*c* >= 'A' && *c* <= 'Z') to the check that $0 \leq c - 'A' \leq 'Z' - 'A'$. The compiler can perform this check using a single unsigned comparison.

Unfortunately, the *SUB* instruction uses the value of *c* directly after the *LDRB* instruction that loads *c*. Consequently, the *ARM9TDMI* pipeline will stall for two cycles. The compiler can't do any better since everything following the load of *c* depends on its value.

However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly. We call these methods load scheduling by *preloading* and *unrolling*.

» *Load Scheduling by Preloading & Load Scheduling by Unrolling – Self Study.*

REGISTER ALLOCATION:

You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the *stack pointer*, *r13*, and the *program counter*, *r15*. For a function to be *ATPCS* compliant it must preserve the callee values of registers *r4* to *r11*. *ATPCS* also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:

```

routine_name
    STMFD sp!,      {r4-r12, lr}      ; stack saved registers
    ; body of routine
    ; the fourteen registers r0-r12 and lr are available
    LDMFD sp!,      {r4-r12, pc}      ; restore registers and return

```

The only purpose in stacking *r12* is to keep the stack eight-byte aligned. You need not stack *r12* if your routine doesn't call other *ATPCS* routines. For *ARMv5* and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an *ARMv4T* processor, then modify the template as follows:

```

routine_name
    STMFD sp!,      {r4-r12, lr}      ; stack saved registers
    ; body of routine
    ; registers r0-r12 and lr available
    LDMFD sp!,      {r4-r12, lr}      ; restore registers
    BX      lr                      ; return, with mode switch

```

Allocating Variables to Register Numbers:

When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.

For the most part ARM operations are orthogonal with respect to register number. In other words, specific register numbers do not have specific roles. If you swap all occurrences of two registers *Ra* and *Rb* in a routine, the function of the routine does not change.

However, there are several cases where the physical number of the register is important:

- ✓ *Argument registers:* The *ATPCS* convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack.
 - The return value must be placed in *r0*.
- ✓ *Registers used in a load or store multiple:* Load and store multiple instructions *LDM* and *STM* operate on a list of registers in order of ascending register number. If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
- ✓ *Load and store double word:* The *LDRD* and *STRD* instructions introduced in *ARMv5E* operate on a pair of registers with sequential register numbers, *Rd* and *Rd + 1*. Furthermore, *Rd* must be an even register number.

Using More Than 14 Local Variables:

If you need more than 14 local 32-bit variables in a routine, then you must store some variables on the stack. The standard procedure is to work outwards from the innermost loop of the algorithm, since the innermost loop has the greatest performance impact.

Making the Most of Available Registers:

On load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory. There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance.

CONDITIONAL EXECUTION:

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the assembler defaults to execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags *N*, *Z*, *C*, *V* stored in the *cpsr* register.

By default, ARM instructions do not update the *N*, *Z*, *C*, *V* flags in the ARM *cpsr*. For most instructions, to update these flags you append an *S* suffix to the instruction mnemonic.

Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the *S* suffix.

By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

Example 17: The following *C* code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character *c*:

<pre>if (i<10) { c = i + '0'; } else { c = i + 'A'-10; }</pre>	<p>We can write this in assembly using conditional execution rather than conditional branches:</p>	<pre>CMP i, #10 ADDLO c, i, #'0' ADDHS c, i, #'A'-10</pre>
---	--	--

The sequence works since the first *ADD* does not change the condition codes. The second *ADD* is still conditional on the result of the compare.

Conditional execution is even more powerful for cascading conditions.

Example 18: The following C code identifies if *c* is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
    vowel++;
}
```

In assembly you can write this using conditional comparison:

```
TEQ    c, #'a'
TEQNE  c, #'e'
TEQNE  c, #'i'
TEQNE  c, #'o'
TEQNE  c, #'u'
ADDEQ  vowel, vowel, #1
```

As soon as one of the *TEQ* comparisons detects a match, the Z flag is set in the *cpsr*. The following *TEQNE* instructions have no effect as they are conditional on $Z = 0$. The next instruction to have effect is the *ADDEQ* that increments *vowel*. You can use this method whenever all the comparisons in the if statement are of the same type.

Example 19: Consider the following code that detects if *c* is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \leq c \leq \text{limit}$. Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB     temp, c, #'A'
CMP     temp, #'Z'-'A'
SUBHI   temp, c, #'a'
CMPHI   temp, #'z'-'a'
ADDLS   letter, letter, #1
```

Note that the logical operations *AND* and *OR* are related by the standard logical relations as shown in the following Table. You can invert logical expressions involving *OR* to get an expression involving *AND*, which can often be useful in simplifying or rearranging logical expressions.

Inverted expression	Equivalent
<code>!(a && b)</code>	<code>(!a) (!b)</code>
<code>!(a b)</code>	<code>(!a) && (!b)</code>

LOOPING CONSTRUCTS:

Most routines critical to performance will contain a loop. Note that, ARM loops are fastest when they count down towards zero. This section describes how to implement these loops efficiently in assembly. We also look at examples of how to unroll loops for maximum performance.

Decrementing Counted Loops:

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```

        MOV i, N
loop
    ; loop body goes here and i=N,N-1,...,1
    SUBS i, i, #1
    BGT loop

```

The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch. On ARM7 and ARM9 this overhead costs four cycles per loop. If i is an array index, then you may want to count down from $N-1$ to 0 inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```

        SUBS i, N, #1
loop
    ; loop body goes here and i=N-1,N-2,...,0
    SUBS i, i, #1
    BGE loop

```

In this arrangement the Z flag is set on the last iteration of the loop and cleared for other iterations. If there is anything different about the last loop, then we can achieve this using the *EQ* and *NE* conditions. For example, if you preload data for the next loop, then you want to avoid the preload on the last loop. You can make all preload operations conditional on *NE*.

There is no reason why we must decrement by one on each loop. Suppose we require $N/3$ loops; rather than attempting to divide N by three, it is far more efficient to subtract three from the loop counter on each iteration:

```

        MOV i, N
loop
    ; loop body goes here and iterates (round up)(N/3) times
    SUBS i, i, #3
    BGT loop

```

Unrolled Counted Loops:

Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome.

Multiple Nested Loops:

How many loop counters does it take to maintain multiple nested loops? Actually, one will suffice—or more accurately, one provided the sum of the bits needed for each loop count does not exceed 32. We can combine the loop counts within a single register, placing the innermost loop count at the highest bit positions.

Other Counted Loops:

You may want to use the value of a loop counter as an input to calculations in the loop. It's not always desirable to count down from N to 1 or $N - 1$ to 0 . For example, you may want to select bits out of a data register one at a time; in this case you may want a power-of-two mask that doubles on each iteration.

The following subsections show useful looping structures that count in different patterns. They use only a single instruction combined with a branch to implement the loop.

Negative Indexing: This loop structure counts from $-N$ to 0 (inclusive or exclusive) in steps of size *STEP*.

```

        RSB    i, N, #0        ; i=-N
loop
    ; loop body goes here and i=-N,-N+STEP,...,
    ADDS    i, i, #STEP
    BLT     loop              ; use BLT or BLE to exclude 0 or not

```

Logarithmic Indexing: This loop structure counts down from 2^N to 1 in powers of two. For example, if $N = 4$, then it counts 16, 8, 4, 2, 1.

```

MOV    i, #1
MOV    i, i, LSL N
loop
; loop body
MOVS   i, i, LSR#1
BNE    loop

```

The following loop structure counts down from an N -bit mask to a one-bit mask. For example, if $N = 4$, then it counts 15, 7, 3, 1.

```

MOV    i, #1
RSB    i, i, i, LSL N ; i=(1<<N)-1
loop
; loop body
MOVS   i, i, LSR#1
BNE    loop

```

Module III

C Compilers and optimization

Overview of C Compilers and optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e., CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Optimizing code takes time and reduces source code readability. Usually, it's only worth optimizing functions that are frequently executed and important for performance. It is recommended to use a performance profiling tool, found in most ARM simulators, to find these frequently executed functions.

C compilers must translate your C function literally into assembler so that it works for all possible inputs. In practice, many of the input combinations are not possible or won't occur.

Let's start by looking at an example of the problems the compiler faces.

Example: The memclr function clears N bytes of memory at address data.

```
void memclr(char *data, int N)
{ for (; N>0; N-
-)
{
*data=0; data++;
}
}
```

- No matter how advanced the compiler, it does not know whether N can be 0 on input or not. Therefore, the compiler needs to test for this case explicitly before the first iteration of the loop.
- The compiler doesn't know whether the data array pointer is four-byte aligned or not. If it is four-byte aligned, then the compiler can clear four bytes at a time using an int store rather than a char store.
- Nor does it know whether N is a multiple of four or not. If N is a multiple of four, then the compiler can repeat the loop body four times or store four bytes at a time using an int store.

The compiler must be conservative and assume all possible values for N and all possible alignments for data. To write efficient C code, you must be aware of areas where the C compiler must be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of a specific C compiler.

Basic C Data Types

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture. In other words, you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

- Early versions of the ARM architecture (ARMv1 to ARMv3) provided hardware support for loading and storing unsigned 8-bit and unsigned or signed 32-bit values.

Table 1 shows the load/store instruction classes available by ARM architecture.

- In Table 1 loads that act on 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign extended. This means that the cast of a loaded value to an int type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an int to smaller type does not cost extra instructions on a store.
- The ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions.
- Finally, ARMv5 adds instruction support for 64-bit load and stores. This is available in ARM9E and later cores.
- Prior to ARMv4, ARM processors were not good at handling signed 8-bit or any 16-bit values. Therefore, ARM C compilers define char to be an unsigned 8-bit value, rather than a signed 8-bit value as is typical in many other compilers.

Table 1: Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store a signed or unsigned 32-bit value
ARMv4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value
	STRH	store a signed or unsigned 16-bit value
ARMv5	LDRD	load a signed or unsigned 64-bit value
	STRD	store a signed or unsigned 64-bit value

Table 2: C compiler datatype mappings.

C Data Type	Implementation
char	unsigned 8-bit byte
short	signed 16-bit halfword
int	signed 32-bit word
long	signed 32-bit word
long long	signed 64-bit double word

Local variable types

- ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, int or long, for local variables wherever possible.
- Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value. The one exception is when you want wrap-around to occur (If you require modulo arithmetic of the form $255+1=0$, then use the char type.)

To see the effect of local variable types, let's consider a simple example.

Examples: We'll look in detail at a checksum function that sums the values in a data packet. Most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

1. The following code checksums a data packet containing 64 words. It shows why you should avoid using char for local variables.

```
int checksum_v1(int *data)
{ char i; int sum = 0;
for (i = 0; i < 64;
i++)
{
sum += data[i];
}
return sum; }
```

At first sight it looks as though declaring i as a char is efficient. You may be thinking that a char uses less register space or less space on the ARM stack than an int. On the ARM, both these assumptions are wrong.

All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the i++ exactly, the compiler must account for the case when i = 255. Any attempt to increment 255 should produce the answer 0.

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
checksum_v1
    BCC
    MOV
    MOV r2,r0 ; r2 = data
    MOV r0,#0 ; sum = 0
    MOV r1,#0 ;i=0
checksum_v1_loop
    LDR r3,[r2,r1,LSL #2] ; r3 = data[i]
    ADD r1,r1,#1 ; r1 = i+1
    AND r1,r1,#0xff ; i = (char)r1
    CMP r1,#0x40 ; compare i, 64
    ADD r0,r3,r0 ; sum += r3
```

```
checksum_v1_loop ; if (i<64) loop
    pc,r14 ; return sum
```

2. Now compare this to the compiler output where instead we declare i as an unsigned int.

```
checksum_v2
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v2_loop
LDR r3,[r2,r1,LSL #2] ; r3 = data[i]
ADD r1,r1,#1 ; r1++
CMP r1,#0x40 ; compare i, 64
ADD r0,r3,r0 ; sum += r3
BCC checksum_v2_loop ; if (i<64) goto loop
MOV pc,r14 ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce i to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

3. Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum.

```
short checksum_v3(short *data)
{ unsigned int i;
  short sum = 0; for (i
= 0; i < 64; i++)
  {
    sum = (short)(sum + data[i]);
  }
  return sum; }
```

You may wonder why the for loop body doesn't contain the code: `sum += data[i];`

The expression `sum + data[i]` is an integer and so can only be assigned to a short using an (implicit or explicit) narrowing cast. As you can see in the following assembly output, the compiler must insert extra instructions to implement the narrowing cast:

```
checksum_v3
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v3_loop
ADD r3,r2,r1,LSL #1 ; r3 = &data[i]
LDRH r3,[r3,#0] ; r3 = data[i]
ADD r1,r1,#1 ; i++
CMP r1,#0x40 ; compare i, 64
ADD r0,r3,r0; r0 = sum + r3
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16 ; sum = (short)r0
checksum_v3_loop ; if (i<64) goto loop
```

```
pc,r14 ; return sum
```

The loop is now three instructions longer than the loop for example `checksum_v2` earlier. There are two reasons for the extra instructions:

- The `LDRH` instruction does not allow for a shifted address offset as the `LDR` instruction did in `checksum_v2`. Therefore, the first `ADD` in the loop calculates the address of item `i` in the array. The `LDRH` loads from an address with no offset.
- The cast reducing `total + array[i]` to a short requires two `MOV` instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend. The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

We can avoid the second problem by using an `int` type variable to hold the partial sum. We only reduce the sum to a short type at the function exit.

However, the first problem is a new issue. We can solve it by accessing the array by incrementing the pointer `data` rather than using an index as in `data[i]`. This is efficient regardless of array type size or element size. All ARM load and store instructions have a post increment addressing mode.

4. The `checksum_v4` code fixes all the problems we have discussed in this section. It uses `int` type local variables to avoid unnecessary casts. It increments the pointer `data` instead of using an index offset `data[i]`.

```
short checksum_v4(short *data)
{ unsigned int i; int
sum=0; for (i=0;
i<64; i++)
{
sum += *(data++);
} return
(short)sum; }
```

The `*(data++)` operation translates to a single ARM instruction that loads the data and increments the data pointer. Of course, you could write `sum += *data; data++;` or even `*data++` instead if you prefer.

The compiler produces the following output. Three instructions have been removed from the inside loop, saving three cycles per loop compared to `checksum_v3`.

```
checksum_v4
MOV r2,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v4_loop
LDRSH r3,[r0],#2 ; r3 = *(data++)
ADD r1,r1,#1 ; i++
CMP r1,#0x40 ; compare i, 64
ADD r2,r3,r2 ; sum += r3
checksum_v4_loop; if (sum<64) goto loop
r0,r2,LSL #16
```

```
MOV r0,r0,ASR #16 ; r0 = (short)sum
MOV pc,r14 ; return r0
```

The compiler is still performing one cast to a 16-bit range, on the function return. You could remove this also by returning an int result.

Function Argument types

We know that converting local variables from types char or short to type int increases performance and reduces code size. The same holds for function arguments.

Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{ return a+ (b >>
1); }
```

This function is useful test case to illustrate the problems faced by the compiler. The input values a, b, and the return value will be passed in 32-bit ARM registers.

Should the compiler assume that these 32-bit values are in the range of a short type, that is, 32,768 to 32,767? Or should the compiler force values to be in this range by sign-extending the lowest 16 bits to fill the 32-bit register? The compiler must make compatible decisions for the function caller and callee. Either the caller or callee must perform the cast to a short type.

We say that function arguments are passed wide if they are not reduced to the range of the type and narrow if they are.

If the compiler

- passes arguments wide, then the callee must reduce function arguments to the correct range.
- passes arguments narrow, then the caller must reduce the range.
- returns values wide, then the caller must reduce the return value to the correct range.
- returns values narrow, then the callee must reduce the range before returning the value.

The armcc output for add_v1 shows that the compiler casts the return value to a short type but does not cast the input values. It assumes that the caller has already ensured that the 32-bit values r0 and r1 are in the range of the short type. This shows narrow passing of arguments and return value.

```
add_v1
ADD r0,r0,r1,ASR #1 ; r0 = (int)a + ((int)b >> 1)
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16 ; r0 = (short)r0
MOV pc,r14 ; return r0
```

The gcc compiler we used is more cautious and makes no assumptions about the range of argument value. This version of the compiler reduces the input arguments to the range of a short in both the caller and the callee. It also casts the return value to a short type. Here is the compiled code for add_v1:

```

add_v1_gcc
MOV r0, r0, LSL #16
MOV r1, r1, LSL #16
MOV r1, r1, ASR #17 ; r1 = (int)b >> 1
ADD r1, r1, r0, ASR #16 ; r1 += (int)a
MOV r1, r1, LSL #16
MOV r0, r1, ASR #16 ; r0 = (short)r1
MOV pc, lr ; return r0

```

Whatever the merits of different narrow and wide calling protocols, you can see that char or short type function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the int type for function arguments and return values, even if you are only passing an 8-bit value.

Signed versus Unsigned Types

The previous sections demonstrate the advantages of using int rather than a char or short type for local variables and function arguments. This section compares the efficiencies of signed int and unsigned int.

- If your code uses addition, subtraction, and multiplication, then there is no performance difference between signed and unsigned operations.
- However, there is a difference when it comes to division. Consider the following short example that averages two integers:

```

int average_v1(int a, int b)
{
    return (a+b)/2;
} This
compiles to
average_v1
ADD r0,r0,r1 ; r0=a+b
ADD r0,r0,r0,LSR #31 ; if (r0<0) r0++
MOV r0,r0,ASR #1 ; r0 = r0 >> 1
MOV pc,r14 ; return r0

```

Notice that the compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces $x/2$ by the statement: $(x<0) ? ((x+1) >> 1) : (x >> 1)$

It must do this because x is signed. In C on an ARM target, a divide by two is not a right shift if x is negative. For example, $-3 >> 1 = -2$ but $-3/2 = -1$. Division rounds towards zero, but arithmetic right shift rounds towards -infinity.

It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types.

SUMMARY: The Efficient Use of C Types

- For local variables held in registers, don't use a char or short type unless 8-bit or 16-bit modular arithmetic is necessary. Use the signed or unsigned int types instead. Unsigned types are faster when you use divisions.
- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint. The ARMv4 architecture is efficient at loading and storing all data widths provided you traverse arrays by incrementing the array pointer. Avoid using offsets from the base of the array with short type arrays, as LDRH does not support this.
- Use explicit casts when reading array entries or global variables into local variables or writing local variables out to array entries.

The casts make it clear that for fast operation you are taking a narrow width type stored in memory and expanding it to a wider type in the registers. Switch on implicit narrowing cast warnings in the compiler to detect implicit casts.

- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles. Casts on loads or stores are usually free because the load or store instruction performs the cast for you.
- Avoid char and short types for function arguments or return values. Instead use the int type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts.

C Looping Structures

Loops with a fixed number of iterations

What is the most efficient way to write a for loop on the ARM? Let's return to our checksum example and look at the looping structure. Here is the last version of the 64-word packet checksum routine. This shows how the compiler treats a loop with incrementing count `i++`.

```
int checksum_v5(int *data)
{ unsigned int i; int
sum=0; for (i=0; i<64;
i++)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v5
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v5_loop
LDR r3,[r2],#4 ; r3 = *(data++) ADD
r1,r1,#1 ; i++
CMP r1,#0x40 ; compare i, 64
```

```
ADD r0,r3,r0; sum += r3
BCC checksum_v5_loop ; if (i<64) goto loop
MOV pc,r14 ; return sum
```

It takes three instructions to implement the for loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i < 64

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result.
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags.

Example:

1. shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{ unsigned int i; int
sum=0; for (i=64; i!=0; i-
-)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v6
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0x40 ;i= 64
checksum_v6_loop
LDR r3,[r2],#4 ; r3 = *(data++)
SUBS r1,r1,#1 ; i-- and set flags
ADD r0,r3,r0 ; sum += r3
BNE checksum_v6_loop ; if (i!=0) goto loop
MOV pc,r14 ; return sum
```

The SUBS and BNE instructions implement the loop. Our checksum example now has the minimum number of four instructions per loop. This is much better than six for checksum_v1 and eight for checksum_v3.

For an unsigned loop counter i we can use either of the loop continuation conditions $i \neq 0$ or $i > 0$. As i can't be negative, they are the same condition.

For a signed loop counter, it is tempting to use the condition $i > 0$ to continue the loop. You might expect the compiler to generate the following two instructions to implement the loop:

```
SUBS r1,r1,#1 ; compare i with 1, i=i-1
BGT loop ; if (i+1>1) goto loop
```

In fact, the compiler will generate,

```
SUB r1,r1,#1 ; i--
CMP r1,#0 ; compare i with 0
BGT loop ; if (i>0) goto loop
```

For the first piece of code the SUBS instruction compares i with 1 and then decrements i . Since $-0x80000000 < 1$, the loop terminates.

For the second piece of code, we decrement i and then compare with 0. Modulo arithmetic means that i now has the value $+0x7ffffff$, which is greater than zero. Thus, the loop continues for many iterations.

Therefore you should use the termination condition $i \neq 0$ for signed or unsigned loop counters. It saves one instruction over the condition $i > 0$ for signed i .

Loops using a variable number of iterations.

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable N giving the number of words in the data packet. Using the lessons from the last section we count down until $N=0$ and don't require an extra loop counter i .

Example: 1. The checksum_v7 shows how the compiler handles a for loop with a variable number of iterations N .

```
int checksum_v7(int *data, unsigned int N)
{ int sum=0; for
( ; N!=0; N--)
{
sum += *(data++);
}
return sum;
}
This compiles to
checksum_v7
MOV r2,#0 ; sum = 0
CMP r1,#0 ; compare N, 0
BEQ checksum_v7_end; if (N==0) goto end
checksum_v7_loop

LDR r3,[r0],#4 ; r3 = *(data++)
SUBS r1,r1,#1 ; N-- and set flags
ADD r2,r3,r2 ; sum += r3
```

```
BNE checksum_v7_loop; if (N!=0) goto loop
checksum_v7_end
MOV r0,r2 ; r0 = sum
MOV pc,r14 ; return r0
```

Notice that the compiler checks that N is nonzero on entry to the function. Often this check is unnecessary since you know that the array won't be empty. In this case a do-while loop gives better performance and code density than a for loop.

2. This example shows how to use a do-while loop to remove the test for N being zero that occurs in a for loop.

```
int checksum_v8(int *data, unsigned int N)
{
    int
    sum=0; do
    {
        sum += *(data++);
    } while (--N!=0); return sum;
}
```

The compiler output is now

```
checksum_v8
MOV r2,#0; sum = 0
checksum_v8_loop
LDR r3,[r0],#4 ; r3 = *(data++)
SUBS r1,r1,#1 ; N-- and set flags
ADD r2,r3,r2 ; sum += r3
BNE checksum_v8_loop ; if (N!=0) goto loop
MOV r0,r2 ; r0 = sum
MOV pc,r14 ; return r0
```

Compare this with the output for checksum_v7 to see the two-cycle saving.

Loop Unrolling

We saw in previous Section that each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch. We call these instructions the loop overhead.

On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

You can save some of these cycles by unrolling a loop - repeating the loop body several times, and reducing the number of loop iterations by the same proportion.

Example,

1. let's unroll our packet checksum example four times.

The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet N is a multiple of four.

```

int checksum_v9(int *data, unsigned int N)
{
    int sum=0;
    do {
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
        N-= 4;
    } while ( N!=0);
    Return sum;
}

```

This compiles to

```

checksum_v9
MOV r2,#0 ; sum = 0
checksum_v9_loop
LDR r3,[r0],#4 ; r3 = *(data++)
SUBS r1,r1,#4 ; N -= 4 & set flags
ADD r2,r3,r2 ;sum += r3
LDR r3,[r0],#4 ;r3 = *(data++)
ADD r2,r3,r2 ;sum += r3
LDR r3,[r0],#4 ;r3 = *(data++)
ADD r2,r3,r2 ;sum += r3
LDR r3,[r0],#4 ;r3 = *(data++)
ADD r2,r3,r2 ;sum += r3
BNE checksum_v9_loop ;if (N!=0) goto loop
MOV r0,r2 ;r0 = sum
MOV pc,r14 ;return r0

```

loop overhead has been reduced the from 4N cycles to (4N)/4=N cycles.

On the ARM7TDMI, this accelerates the loop from 8 cycles per accumulate to 20/4=5 cycles per accumulate, nearly doubling the speed!

There are two questions you need to ask when unrolling a loop:

- How many times should I unroll the loop?
- What if the number of loop iterations is not a multiple of the unroll amount? For example, what if N is not a multiple of four in checksum_v9?

To start with the first question, only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.

For the second question, try to arrange it so that array sizes are multiples of unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases. This increases the code size a little but keeps the performance high.

2. This example handles the checksum of any size of data packet using a loop that has been unrolled four times.

```
int checksum_v10(int *data, unsigned int N)
{ unsigned int i; int
sum=0; for (i=N/4; i!=0; i--)
{
sum += *(data++);
sum+= *(data++);
sum +=*(data++);
sum +=*(data++);
} for (i=N&3; i!=0; i--)
{
sum += *(data++);
}
return sum; }
```

The second for loop handles the remaining cases when N is not a multiple of four. Note that both N/4 and N&3 can be zero, so we can't use do-while loops.

SUMMARY: Writing Loops Efficiently

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free.
- Use unsigned loop counters by default and the continuation condition `i!=0` rather than `i>0`. This will ensure that the loop overhead is only two instructions.
- Use do-while loops rather than for loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero.
- Unroll important loops to reduce the loop overhead. Do not over unroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache.
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loops easily by two, four, or eight times without worrying about the leftover array elements.

Register Allocation

The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables does not overlap.

When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped out variables since they are written out to memory. Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

First let's look at the number of processor registers the ARM C compilers have available for allocating variables. Table 3 shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

Table 3: C compiler register usage.

Register number	Alternate register names	ATPCS register usage
<i>r0</i>	<i>a1</i>	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
<i>r1</i>	<i>a2</i>	
<i>r2</i>	<i>a3</i>	
<i>r3</i>	<i>a4</i>	
<i>r4</i>	<i>v1</i>	General variable registers. The function must preserve the callee values of these registers.
<i>r5</i>	<i>v2</i>	
<i>r6</i>	<i>v3</i>	
<i>r7</i>	<i>v4</i>	
<i>r8</i>	<i>v5</i>	
<i>r9</i>	<i>v6 sb</i>	General variable register. The function must preserve the callee value of this register except when compiling for <i>read-write position independence</i> (RWPI). Then <i>r9</i> holds the <i>static base</i> address. This is the address of the read-write data.
<i>r10</i>	<i>v7 sl</i>	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then <i>r10</i> holds the stack limit address.
<i>r11</i>	<i>v8 fp</i>	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of <i>armcc</i> use a frame pointer.
<i>r12</i>	<i>ip</i>	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
<i>r13</i>	<i>sp</i>	The stack pointer, pointing to the full descending stack.
<i>r14</i>	<i>lr</i>	The link register. On a function call this holds the return address.
<i>r15</i>	<i>pc</i>	The program counter.

Provided the compiler is not using software stack checking, then the C compiler can use registers *r0* to *r12* and *r14* to hold variables. It must save the callee values of *r4* to *r11* and *r14* on the stack if using these registers.

In theory, the C compiler can assign 14 variables to registers without spillage. In practice, some compilers use a fixed register such as *r12* for intermediate scratch working and do not assign variables to this register. Also, complex expressions require intermediate working registers to

evaluate. Therefore, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.

If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. A variable used inside a loop counts multiple times. You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

The register keyword in C hints that a compiler should allocate the given variable to a register. However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM). Therefore, we recommend that you avoid using register and rely on the compiler's normal register allocation routine.

SUMMARY: Efficient Register Allocation

- Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.
- You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

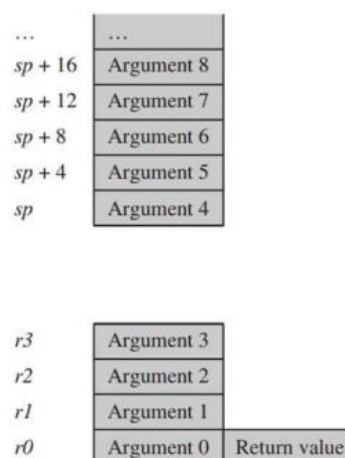
Function calls

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers.

The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: r0, r1, r2, and r3. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 1. Function return integer values are passed in r0.

Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in r0, r1. The compiler may pass structures in registers or by reference according to command line compiler options.



ATPCS argument passing.

The first point to note about the procedure call standard is the four-register rule. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments.

- For functions with four or fewer arguments, the compiler can pass all the arguments in registers.
- For functions with more arguments, both the caller and callee must access the stack for some arguments.

If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than multiple arguments.

Example: 1. illustrates the benefits of using a structure pointer. First, we show a typical routine to insert N bytes from array data into a queue. We implement the queue using a cyclic buffer with start address Q_start (inclusive) and end address Q_end (exclusive).

```
char *queue_bytes_v1( char *Q_start, /* Queue
buffer start address */ char *Q_end, /* Queue
buffer end address */ char *Q_ptr, /* Current
queue pointer position */ char *data, /* Data to
insert into the queue */ unsigned int N) /* Number of bytes
to insert */
{ do
{
*(Q_ptr++) = *(data++); if
(Q_ptr == Q_end)
{
Q_ptr = Q_start;
}
} while (--N); return
Q_ptr;
}
```

This compiles to

```
queue_bytes_v1
STR r14,[r13,#-4]! ; save lr on the stack
LDR r12,[r13,#4] ; r12 = N
queue_v1_loop
LDRB r14,[r3],#1 ; r14 = *(data++)
STRB r14,[r2],#1 ; *(Q_ptr++) = r14
CMP r2,r1 ; if (Q_ptr == Q_end)
MOVEQ r2,r0 ; {Q_ptr = Q_start;}
SUBS r12,r12,#1 ; --N and set flags
BNE queue_v1_loop ; if (N!=0) goto loop
MOV r0,r2 ; r0 = Q_ptr
LDR pc,[r13],#4 ; return r0
```

Compare this with a more structured approach using three function arguments.

2. The following code creates a Queue structure and passes this to the function to reduce the number of function arguments.

```
typedef struct {
char *Q_start; /* Queue buffer start address */
char *Q_end; /* Queue buffer end address */
char *Q_ptr;
} Queue;
/* Current queue pointer position */
void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
char *Q_ptr = queue->Q_ptr; char *Q_end = queue->Q_end; do
{
*(Q_ptr++) = *(data++); if
(Q_ptr == Q_end)
{
Q_ptr = queue->Q_start;
}
} while (--N); queue->Q_ptr
= Q_ptr;
}
```

This compiles to

```
queue_bytes_v2
STR r14,[r13,#-4]! ; save lr on the stack
LDR r3,[r0,#8] ; r3 = queue->Q_ptr
LDR r14,[r0,#4] ; r14 = queue->Q_end queue_v2_loop
LDRB r12,[r1],#1 ; r12 = *(data++) STRB r12,[r3],#1 ;
*(Q_ptr++) = r12
CMP r3,r14 ; if (Q_ptr == Q_end) LDREQ
r3,[r0,#0] ; Q_ptr = queue->Q_start SUBS
r2,r2,#1 ; --N and set flags
BNE queue_v2_loop ; if (N!=0) goto loop
STR r3,[r0,#8] ; queue->Q_ptr = r3
LDR pc,[r13],#4 ; return
```

The queue_bytes_v2 is one instruction longer than queue_bytes_v1, but it is in fact more efficient overall.

The second version has only three function arguments rather than five. Each call to the function requires only three register setups. This compares with four register setups, a stack push, and a stack pull for the first version. There is a net saving of two instructions in function call overhead.

There are other ways of reducing function call overhead if your function is very small and corrupts few registers (uses few local variables). Put the C function in the same C file as the functions that will call it. The C compiler then knows the code generated for the callee function and can make optimizations in the caller function:

- The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore, the caller function need not save all the ATPCS corruptible registers.
- If the callee function is very small, then the compiler can inline the code in the caller function. This removes the function call overhead completely.

3. The function `uint_to_hex` converts a 32-bit unsigned integer into an array of eight hexadecimal digits. It uses a helper function `nybble_to_hex`, which converts a digit `d` in the range 0 to 15 to a hexadecimal digit.

```
unsigned int nybble_to_hex(unsigned int d)
{ if (d<10) {
return d + '0';
} return d- 10+
'A';
}

void uint_to_hex(char *out, unsigned int in)
{ unsigned int i;
for (i=8; i!=0; i--)
{
in = (in << 4) | (in >> 28); /* rotate in left by 4 bits */
*(out++) = (char)nybble_to_hex(in & 15);
}
}
```

When we compile this, we see that `uint_to_hex` doesn't call `nybble_to_hex` at all! In the following compiled code, the compiler has inlined the `uint_to_hex` code. This is more efficient than generating a function call.

```
uint_to_hex
MOV r3,#8 ;i=8
uint_to_hex_loop
MOV r1,r1,ROR #28 ; in = (in << 4)|(in >> 28)
AND r2,r1,#0xf ; r2 = in & 15
CMP r2,#0xa ; if (r2>=10)
ADDCS r2,r2,#0x37 ; r2 += 'A'-10
ADDCC r2,r2,#0x30 ; else r2 += '0'
STRB r2,[r0],#1 ; *(out++) = r2
SUBS r3,r3,#1 ; i-- and set flags
BNE uint_to_hex_loop ; if (i!=0) goto loop
MOV pc,r14 ; return
```

The compiler will only inline small functions. You can ask the compiler to inline a function using the `__inline` keyword. Inlining large functions can lead to big increases in code size without much performance improvement.

SUMMARY: Calling Functions Efficiently

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.

- Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.
- Critical functions can be inlined using the `__inline` keyword.

Pointer Aliasing

Two pointers are said to alias when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer.

In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Example: The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
    *timer1 += *step;
    *timer2 += *step;
} This
```

compiles to

```
timers_v1
LDR r3,[r0,#0] ; r3 = *timer1
LDR r12,[r2,#0] ; r12 = *step
ADD r3,r3,r12 ; r3 += r12
STR r3,[r0,#0] ; *timer1 = r3
LDR r0,[r1,#0] ; r0 = *timer2
LDR r2,[r2,#0] ; r2 = *step
ADD r0,r0,r2; r0 += r2
STR r0,[r1,#0] ; *timer2 = r0
MOV pc,r14 ; return
```

Note that the compiler loads from `step` twice. Usually, a compiler optimization called common subexpression elimination would kick in so that `*step` was only evaluated once, and the value reused for the second occurrence. However, the compiler can't use this optimization here. The pointers `timer1` and `step` might alias one another. In other words, the compiler cannot be sure that the write to `timer1` doesn't affect the read from `step`.

In this case the second value of `*step` is different from the first and has the value `*timer1`. This forces the compiler to insert an extra load instruction.

The same problem occurs if you use structure accesses rather than direct pointer access. The following code also compiles inefficiently:

```
typedef struct {int step;} State; typedef
struct {int timer1, timer2;} Timers; void
timers_v2(State *state, Timers *timers)
{
    timers->timer1 += state->step; timers->timer2
    += state->step;
```

```
}
```

The compiler evaluates `state->step` twice in case `state->step` and `timers->timer1` are at the same memory address. The fix is easy: Create a new local variable to hold the value of `state->step` so the compiler only performs a single load.

In the code for `timers_v3` we use a local variable `step` to hold the value of `state->step`. Now the compiler does not need to worry that `state` may alias with `timers`.

```
void timers_v3(State *state, Timers *timers)
{
    int step = state->step;
    timers->timer1 += step;
    timers->timer2 += step; }
```

You must also be careful of other, less obvious situations where aliasing may occur. When you call another function, this function may alter the state of memory and so change the values of any expressions involving memory reads. The compiler will evaluate the expressions again.

Another pitfall is to take the address of a local variable. Once you do this, the variable is referenced by a pointer and so aliasing can occur with other pointers. The compiler is likely to keep reading the variable from the stack in case aliasing occurs. Consider the following example, which reads and then checksums a data packet:

```
int checksum_next_packet(void)
{ int *data; int N, sum=0;
  data = get_next_packet(&N);
  do {
    sum += *(data++);
  } while (--N);
  return sum; }
```

Here `get_next_packet` is a function returning the address and size of the next data packet. The previous code compiles to `checksum_next_packet`

```
STMFD r13!,{r4,r14} ; save r4, lr on the stack
SUB r13,r13,#8 ; create two stacked variables
ADD r0,r13,#4 ; r0 = &N, N stacked
MOV r4,#0 ; sum = 0
BL get_next_packet ; r0 = data checksum_loop
LDR r1,[r0],#4 ; r1 = *(data++)
ADD r4,r1,r4; sum += r1
LDR r1,[r13,#4] ; r1 = N (read from stack)
SUBS r1,r1,#1 ; r1-- & set flags
STR r1,[r13,#4] ;N= r1 (write to stack)
BNE checksum_loop ; if (N!=0) goto loop
MOV r0,r4 ; r0 = sum
ADD r13,r13,#8 ; delete stacked variables
LDMFD r13!,{r4,pc} ; return r0
```

Note how the compiler reads and writes N from the stack for every N--. Once you take the address of N and pass it to `get_next_packet`, the compiler needs to worry about aliasing because the pointers data and `&N` may alias. To avoid this, don't take the address of local variables. If you must do this, then copy the value into another local variable before use.

SUMMARY: Avoiding Pointer Aliasing

- Do not rely on the compiler to eliminate common subexpressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once.
- Avoid taking the address of local variables. The variable may be inefficient to access from then on.