Getting Better Object Code

Using the ISA Features

How Can We Use These Great Instructions?

- Write **C code, rely on the compiler** to generate instructions
  - Depends on compiler's ability to optimize code
  - Can turn into game of "How can I get the compiler to do what I want?"
- Write **C code with compiler intrinsics** to specify key instructions
  - Provides some control
  - Takes care of most details
- Write **C code with inline assembly code** to specify key instructions
  - Provides more control
  - Takes care of a few details
- Write a separate **assembly code** module, link it with our C code
  - Provides full control
  - You must manage all the details

Do Instruction Set Architectures Matter?

- Online discussion by Jack Ganssle, Bill Gatliff, Niall Murphy, and Jim Turley at Embedded.com

- No!
  - C compiler hides differences and emulates missing features with code
    - Native word size, floating point math, subroutine call penalty, conditional branch delay
  - Most compilers don't ***use*** those great fast instructions
    - Table lookup and interpolate, 3D matrix operations, etc
  - As long as the processor runs fast enough, costs dominate

- Yes!
  - Use "intrinsics" (inline assembly code) to use fast instructions
  - Code density depends on processor
  - It takes time and money to come up to speed on a new architecture, so go with what gets you a product sooner
  - More engineers available for hiring if you use a common architecture
  - If no good tools are available, it's not worth using

How Can We Use These Great Instructions?

- Write **C code, rely on the compiler** to generate instructions
- Write **C code with compiler intrinsics** to specify key instructions
- Write **C code with inline assembly code** to specify key instructions
- Write a separate **assembly code** module, link it with our C code

How to Help the Compiler

Cortex A-Series Programmer's Guide, Chapter 17

- Eliminating common sub-expressions
  - May need to do manually
- Loop unrolling
  - Benefit reduced on A72 thanks to branch prediction and out-of-order execution, but may still be helpful. Depends on your code!
  - May increase cache pressure, reducing performance
- GCC optimization options to use
  - -O3
  - -Ofast
  - -funroll-loops
  - -mcpu=cortex-a72
  - -mfpu=crypto-neon-fp-armv8 *(neon is name for Advanced SIMD technology)*
  - -mfloat-abi=hard
- Loop termination
  - Integer loops counting down to zero get free test (no CMP instruction needed)
  - Use 32-bit int loop counters (native word size)
- Reduce stack and heap usage
  - Minimize live variables
  - Limit function parameters to fit in four registers
- Variable selection
  - Words are faster than bytes and halfwords due to code for extension and overflow

More Help

Cortex A-Series Programmer's Guide, Chapter 17

- Pointer aliasing
  - In general case, pointers can alias (point to overlapping regions in memory), constraining some compiler optimizations
  - Tell compiler pointer parameters can't alias with **restrict** keyword

```
void foo(unsigned int *restrict ptr1,
```

- Division and modulo
  - Division is always slower than multiplication, so replace if possible
  - Modulo is also slow, so avoid it
- Extern data
  - Loading external variables requires extra instructions, so group them in a structure to share the base pointer
- Inline or embedded assembler
  - Avoid it! First try improving algorithm, then compiler optimizations
  - Use –S –fverbose-asm to generate annotated assembly code
- Unaligned access
  - Accesses to non-word-aligned values are slow
  - Accesses crossing cache line boundaries are even slower
  - Turned on with –O2 and above
    - -falign-[functions|labels|loops|jumps]

GCC Compiler Optimization Settings

- Get help from gcc
  - gcc --help=common
  - gcc --help=optimizers
  - gcc -dumpmachine
- Options that control optimizations
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options
- Many tunable parameters
  - --param name=value
  - --help=param -Q
- Target-specific help
  - gcc --target-help
  - https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html
- Common options
  - -O3 optimization (maximum)
  - Optimize for speed, disregarding exact standards compliance
    - -Ofast
    - -ffast-math
    - -funsafe-math-optimizations

Example: Switch Statement for Cosine Range Reduction

```
switch (quad){
case 0: return  cos_32s(x);
case 1: return -cos_32s(DP_PI-x);
case 2: return -cos_32s(x-DP_PI);
case 3: return  cos_32s(twopi_f-x);
}
```

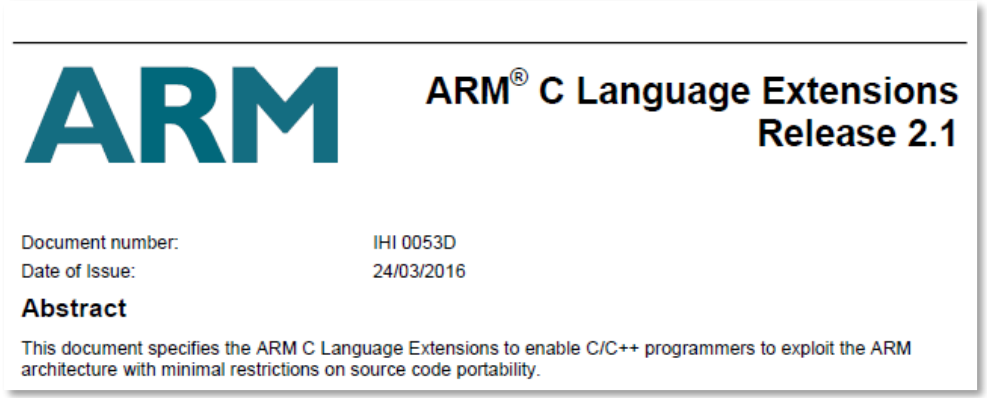
- Switch can be implemented with if/elseif ladder, a jump table, or computed jump…
- Here compiler uses two instructions and a jump table
  - cmp r3,#3 Compare r3 with #3, setting condition flags (NZVC)
  - ldrls pc, …. Conditionally load PC with case code address if r3 is less than or same (<=) as #3
    - Case code addresses stored in jump table starting at 1108c (cases 0, 1, 2, 3)
    - Table location determined by adding pc and r3 left-shifted by 2 (= pc+4*r3)
    - If r3 > 3, then ldrls does nothing, and b11154 executes to branch past last case
- Note how ldrls dominates execution time (>50%)
  - Floating point math executes very quickly
  - Load to PC (if executed) introduces branch delay penalty, flushing all following instructions in pipeline

```
 1.28 11080:   cmp    r3, #3
54.89 11084:   ldrls  pc, [pc, r3, lsl #2]
      11088: ↓ b      11154 <cos_52+0x114>
      1108c:   .word  0x00011130
      11090:   .word  0x000110fc
      11094:   .word  0x000110c8
      11098:   .word  0x0001109c
      1109c:   vldr   s3, [pc, #208]  ; 111
      110a0:   vldr   s12, [pc, #208] ; 111
      110a4:   vldr   s13, [pc, #208] ; 111
      110a8:   vsub.f32 s4, s3, s0
      110ac:   vldr   s14, [pc, #204] ; 111
      110b0:   vldr   s0, [pc, #204]  ; 111
      110b4:   vmul.f32 s5, s4, s4
      110b8:   vfma.f32 s13, s5, s12
      110bc:   vfma.f32 s14, s5, s13
      110c0:   vfma.f32 s0, s5, s14
      110c4: → bx     lr
 1.02 110c8:   vldr   d3, [pc, #152]  ; 111
      110cc:   vldr   s8, [pc, #164]  ; 111
 0.07 110d0:   vldr   s9, [pc, #164]  ; 111
      110d4:   vsub.f64 d5, d16, d3
 0.15 110d8:   vldr   s7, [pc, #160]  ; 111
 0.07 110dc:   vldr   s0, [pc, #160]  ; 111
      110e0:   vcvt.f32.f64 s11, d5
      110e4:   vmul.f32 s15, s11, s11
      110e8:   vfma.f32 s9, s15, s8
      110ec:   vfma.f32 s7, s15, s9
      110f0:   vfma.f32 s0, s15, s7
      110f4:   vneg.f32 s0, s0
      110f8: → bx     lr
```

How Can We Use These Great Instructions?

- Write **C code, rely on the compiler** to generate instructions
- Write **C code with compiler intrinsics** to specify key instructions
- Write **C code with inline assembly code** to specify key instructions
- Write a separate **assembly code** module, link it with our C code

ARM C Language Extensions (ACLE)


ARM

ARM® C Language Extensions
Release 2.1

Document number: IHI 0053D
Date of Issue: 24/03/2016

Abstract

This document specifies the ARM C Language Extensions to enable C/C++ programmers to exploit the ARM architecture with minimal restrictions on source code portability.

- What are intrinsics?
  - Compiler keywords which specify architecture-specific operations: special instructions, support operations
  - May be implemented as functions (__x), macros (__ARM_x), other
- Where are they described?
  - ARM: IHI0053D_acle_2_0.pdf, https://static.docs.arm.com/ihi0053/d/IHI0053D_acle_2_1.pdf
- Note that GCC also provides C Language Extensions, though not ARM-specific:
  - https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html#C-Extensions
- How can we use them?
  - #include <arm_acle.h>
- What can they do?
  - Test for feature support
  - Access special registers (system and coprocessor)
  - Define special attributes
  - Support multiprocessing with synchronization, barriers
  - Prefetch instructions or data
  - Process data
    - Access APSR flags, apply special instructions (rotate, count leading zeros/sign bits, reverse bits/bytes)
    - Saturating math, accumulating multiplies
    - Floating point square root, fused multiply/add
    - 32-bit SIMD: 8-, 16-bit data types and operations (add, multiply, pack, select, sum of absolute differences, etc.)
    - CRC32

Example

9.4.3 Accumulating multiplications

These intrinsics are available when `__ARM_FEATURE_DSP` is defined.

```
int32_t __smlabb(int32_t, int32_t, int32_t);
```

Multiplies two 16-bit signed integers, the low halfwords of the first two operands, and adds to the third operand. Sets the Q flag if the addition overflows. (Note that the addition is the usual 32-bit modulo addition which wraps on overflow, not a saturating addition. The multiplication cannot overflow.)

```
int32_t __smlabt(int32_t, int32_t, int32_t);
```

Multiplies the low halfword of the first operand and the high halfword of the second operand, and adds to the third operand, as for `__smlabb`.

```
int32_t __smlatb(int32_t, int32_t, int32_t);
```

Multiplies the high halfword of the first operand and the low halfword of the second operand, and adds to the third operand, as for `__smlabb`.

How Can We Use These Great Instructions?

- Write **C code, rely on the compiler** to generate instructions
- Write **C code with compiler intrinsics** to specify key instructions
- Write **C code with inline assembly code** to specify key instructions
- Write a separate **assembly code** module, link it with our C code

INLINE ASSEMBLY CODE

Overview

https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C

• Basic Asm:		Inline assembler without operands.
• Extended Asm:		Inline assembler with operands.
• Constraints:		Constraints for `asm` operands
• Asm Labels:		Specifying the assembler name to use for a C symbol.
• Explicit Register Variables:		Defining variables residing in specified registers.
• Size of an asm:		How GCC calculates the size of an `asm` block.

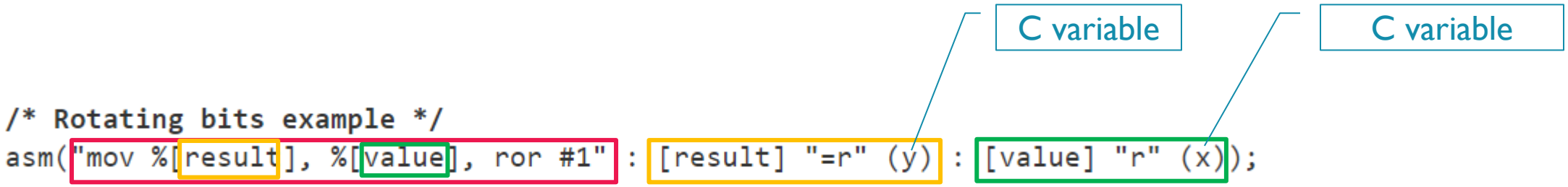
- Use asm (or __asm__) extension keyword
- Syntax for Extended Asm
  - [optional]
  - Qualifiers
    - If code has side-effects, use **volatile** to disable some optimizations
    - If code may jump to a label, use **goto**
  - Parameters
    - **AssemblerTemplate** is fixed text and tokens referring to other parameters
    - **OutputOperands** is list of C variables which are modified
    - **InputOperands** is list of C expressions which are read
    - **Clobbers** is a list of registers or values which are changed (besides **OutputOperands**)
    - **GotoLabels** is list of C labels for possible jump targets

asm *asm-qualifiers* (*AssemblerTemplate*
: *OutputOperands*
[: *InputOperands*
[: *Clobbers*]])

asm *asm-qualifiers* (*AssemblerTemplate*
: *OutputOperands*
: *InputOperands*
: *Clobbers*
: *GotoLabels*)

ARM GCC Inline Assembler Cookbook

http://www.ethernut.de/en/documents/arm-inline-asm.html



- Nice walk-through with explanations on linked webpage
  - Take input integer variable x,
  - Rotate right by one,
  - Write result to another integer variable y
- Parts
  - Assembler Template is string literal ("format string")
  - Output operands: [symbolic name] constraint (C expression)
  - Input operands: [symbolic name] constraint (C expression)
  - Clobber list – *none used here*

ARM GCC Inline Assembler Cookbook

http://www.ethernut.de/en/documents/arm-inline-asm.html

```
asm("mov    %[result], %[value], ror #1"

    : [result]"=r" (y) /* Rotation result. */
    : [value]"r"   (x) /* Rotated value. */
    : /* No clobbers */
);
```

- Alternate form, with all fields.
- Can include comments and white-space

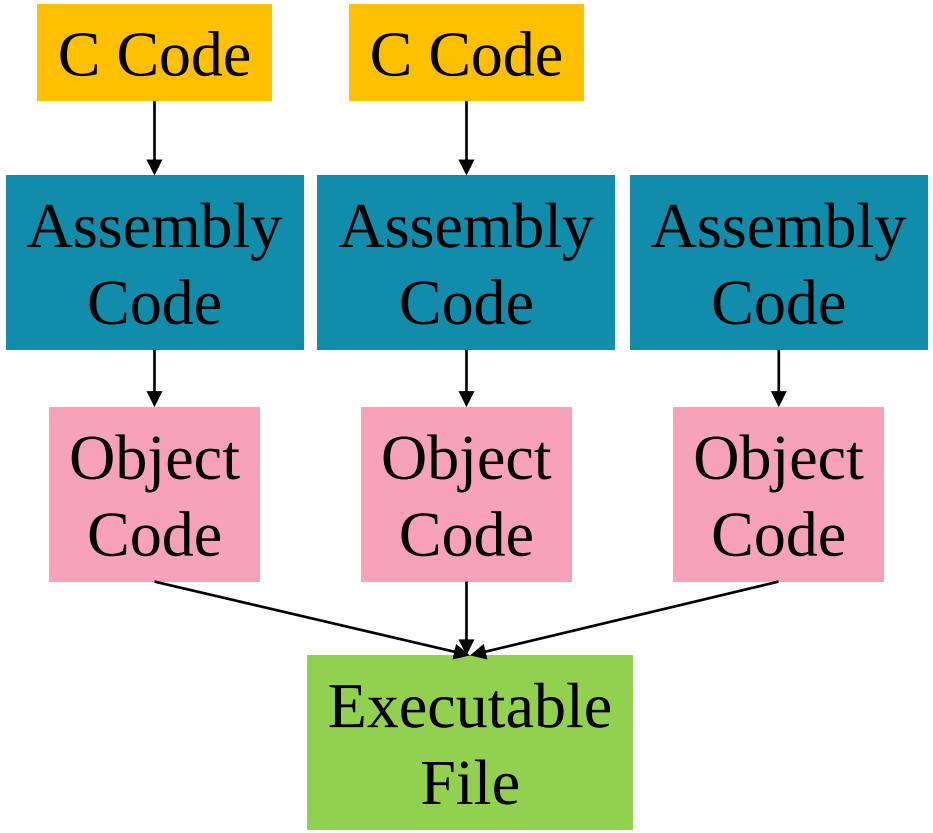
- Parts
  - Assembler Template is string literal ("format string")
  - Output operands: [symbolic name] constraint (C expression)
  - Input operands: [symbolic name] constraint (C expression)
  - Clobber list – *none used here*

How Can We Use These Great Instructions?

- Write **C code, rely on the compiler** to generate instructions
- Write **C code with compiler intrinsics** to specify key instructions
- Write **C code with inline assembly code** to specify key instructions
- Write a separate **assembly code** module, link it with our C code

MIXING C AND ASSEMBLY CODE MODULES

Mixing Modules



- Some modules compiled from C to object code
- Some modules assembled from assembly code to object code
- All linked together into executable

- Assembly code functions must be declared as externs to be visible to C functions
  - Function f defined in assembly code:
    - Arguments: int * , int
    - Return: int
  - C code needs to know ***extern int f(int * a, int b);***

Procedure Call Standard

- Must save and restore certain registers if used
  - ARM: r4-r8, r10-r11
- Follow rules when passing parameters, returning results
- Defined in ARM IHI 0042D, part of Application Binary Interface

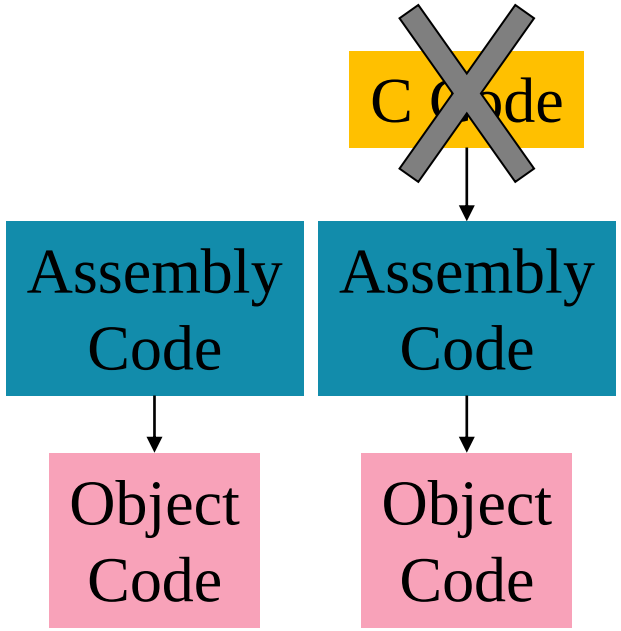
Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Subroutine must restore r10-r11 (v7-v8) to original values before returning

Subroutine must restore r4-r8 (v1-v5) to original values before returning

Table 2, Core registers and AAPCS usage

How Can We Create an Assembly File?



- Two options
  1. Create from scratch, then add assembly code. Hard until you know how to do it.
  2. Create from file with C code, compile to assembly once, remove C file, then modify assembly code

Hard Option: Create Assembly Code from Scratch

```
        .text
        .align 2

        .global      asm_doit
@ This line is a commment
        .type  asm_doit, %function

asm_doit:
        push   {r4, r5, r6, r7}
        movs   r4, #0

.L66:
        adds   r4, r4, #1
        cmp    r4, #4000
        bne    .L66

        pop    {r4, r5, r6, r7}
        bx     lr
```

- .text: executable code will follow
- .align: advance location counter to next address which is multiple of 2^2=4
- .global: makes symbol visible to other modules when linking. Essential for function name!
- .type: specifies symbol is a function
- Label is symbol followed by colon, marks an address
- Function Prologue
  - Push registers which might be modified
    - ARM: r4-r11
    - NEON: Q4-Q7
- Function body
  - Arguments 1, 2, 3, 4 are in r0, r1, r2, r3
- Function Epilogue
  - Place return value (if any) in r0
  - Pop any registers pushed above in prologue
  - bx lr to return from subroutine

Easier Option: Use C Compiler's Assembly Output

- Create separate file foo.c with skeleton C function foo()
  - Include at least arguments and return value
  - Possibly also include function body
- Compile foo.c to assembly source (gcc -S, or make foo.s)
- Move foo.c elsewhere (e.g. into subdirectory) so Make doesn't recompile and link it
- Edit .s file as needed
- Make handles the details

Makefile

```
OBJFILES := $(patsubst %.c,%.o,$(wildcard *.c)) $(patsubst %.s,%.o,$(wildcard *.s))
$(PROJ_NAME): $(OBJFILES)
$(CC) -o $(PROJ_NAME) $(OBJFILES) $(LIBS)
%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<
%.s: %.c
        $(CC) $(CFLAGS) -S -o $@ $<
%.lst: %.c
        $(CC) $(CFLAGS) -Wa,-adhln $(LIBS) $< > $@
clean:
        rm -f *.o *.lst
```

- Makefile changes
  - Add object files from asm code (.s files) to dependency list
  - Add rule to compile but not assemble C source: translate .c to .s
- Implicit rules
  - Translate .s to .o using as