SENG440 Embedded Systems

– Lesson 2: Processors for Embedded Systems –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

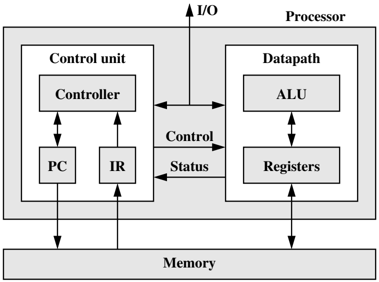
The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 2: Processors for embedded systems

Lesson 2: Course Progress

- **Characteristics and design of embedded systems** – introduction
  - Definition of embedded systems
  - Common characteristics of embedded systems
  - Examples of embedded systems
  - Design process of embedded systems
- **Quality & performance metrics** – measurable features of an embedded system implementation
  - Time-to-market
  - Non-recurring engineering cost and unit cost
  - Performance and flexibility
  - Power consumption
- **Processors for embedded systems**
  - Custom embedded processor design
  - Comparison: ARM vs Power (IBM), 8051 (Intel), 68HC11 (Motorola)
  - Emphasize on **ARM**

General-purpose computing engines – basic elements



- We can identify the constituent elements of the processor: datapath, control path, memory, etc.
- Description of the instruction set will define the architecture

- Think of it in terms of its architecture (instruction set)
- Non-formal description can be useful
  - Description with logic function is impractical
  - Instead, use arithmetic and logic operators
  - Open a handbook of any processor and see what description is used!

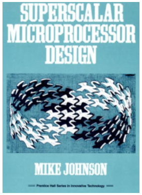
Designing a General-Purpose Processor (GPP)

- This is not something an embedded systems designer normally does
- Major task in GPP design = **design-space exploration**
  - Profile an application (domain) to determine what operations to support in hardware/firmware
  - This is a very complex process
- In embedded systems we do not want to design the control path from the ground up
  - That would trigger major changes in hardware (e.g., memory system, peripherals, etc.) and software (e.g., compiler, debugger, etc.)
- We typically take an existing processor and augment its data path

Recall that...

- For processors we are interested in its architecture:
  - The functions that are provided to the programmer
  - The language for invoking these functions – assembly language
  - The organization of the instruction word
  - Instruction set and addressing modes
- We mostly do not address issues related to the processor's control part

More information:
William M. Johnson,
Superscalar Microprocessors Design,
Prentice Hall 1990.


SUPERSCALAR MICROPROCESSOR DESIGN
MIKE JOHNSON
Prentice Hall Series in Innovative Technology

Migration to embedded domain

- The GPP is intended to solve computation problems in a large variety of applications while the complexity of the processor is kept low
- The (general-purpose) processor is a comodity (unit cost may be very low)
- Do we really need a new processor? Address the design metrics.
  - NO: we have to solve a software design problem
  - YES: we have to solve a hardware-software co-design problem
- Designing a new (embedded) processor:
  - Starting from scratch proves too complicated
  - What is easier and faster to change in the basic scheme? The datapath
- Common strategy: augment the datapath with application-specific units
- There are companies, e.g., Tensilica / Cadence http://www.tensilica.com, that offers templates for different processor classes

General-purpose processors for embedded applications

- Most representative processors in the general-purpose class:

ARM	http://www.arm.com
Power	http://www.power.org
MIPS	http://www.mips.com

- A GPP implementing an embedded application is a **software problem**

  ARM6 in DSP Applications: use of the MUL operation, Application Note 19
  http://www.arm.com/pdfs/DAI0019D_ARM6_DSP.pdf
  Fixed Point Arithmetic on the ARM, Application Note 33
  http://www.arm.com/pdfs/DAI0033A_fixedpoint.pdf
  Writing efficient C for ARM, Application Note 34
  http://www.arm.com/pdfs/DAI0034A_efficient_c.pdf
  Big and Little Endian Byte Addressing, Application Note 61
  http://www.arm.com/pdfs/DAI0061A_byte_addressing.pdf

Migration to embedded domain (cont'd)

- General-Purpose Processors (GPP) augmented with application-specific functional units: **Application-Specific Instruction-set Processors**
- GPPs augmented with standard peripherals
  - Example: I^{2}C interface
  - Topic for project
- GPPs augmented with application-specific coprocessors
  - It resembles a multi-processor system
  - It is outside the course scope
- GPPs augmented with a field-programmable gate array
  - It is a “reconfigurable computing” problem
  - Of increasing interest in the industry

Migration to embedded domain (cont'd)

- General-Purpose Processors (GPP) augmented with application-specific functional units: **Application-Specific Instruction-set Processors**
  Examples:
  - Pentium + Multi-Media eXtension (MMX) http://www.intel.com
  - ARM + DSP capabilities http://www.arm.com/pdfs/ARM-DSP.pdf
  - TMS (digital-signal processor) http://www.ti.com
  - TriMedia (media-oriented processor) http://www.semiconductors.philips.com
- GPPs augmented with standard peripherals
- GPPs augmented with application-specific coprocessors
- GPPs augmented with a field-programmable gate array

Migration to embedded domain (cont'd)

- General-Purpose Processors (GPP) augmented with application-specific functional units: **Application-Specific Instruction-set Processors**
  Examples:
  - Pentium + Multi-Media eXtension (MMX) http://www.intel.com
  - ARM + DSP capabilities http://www.arm.com/pdfs/ARM-DSP.pdf
  - TMS (digital-signal processor) http://www.ti.com
  - TriMedia (media-oriented processor)
    http://www.semiconductors.philips.com
- GPPs augmented with standard peripherals
- GPPs augmented with application-specific coprocessors
- GPPs augmented with a field-programmable gate array

Selecting a processor

- The choice of a processor depends on technical and nontechnical aspects
  - From a technical perspective, one must choose a processor that can achieve the desired speed within certain power, size, and cost constraints
  - Nontechnical aspects may include prior expertise with a processor and its development environment, special licensing arrangements, etc.
- **Speed** is particularly difficult to measure and compare
  - We could compare processor clock speeds, but the number of instructions per clock cycle may differ greatly among processors
  - We could instead compare instructions per second, but the complexity of each instruction may also differ greatly among processors
- We could use a **benchmark** to compare processor performances
  - A benchmark performs no useful work – it focuses on exercising a processor's computation capabilities in order to get statistics
  - There are "lies, damn lies, and statistics" (Mark Twain)

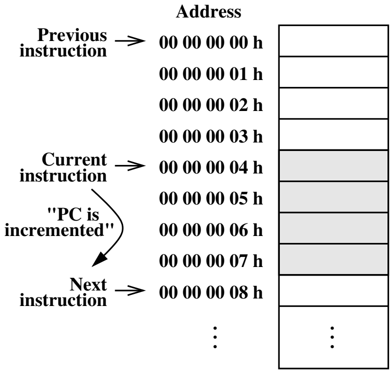
Selecting a processor (cont'd)

- In this heterogenous world, a good strategy is to analyze the processor architecture, that is, **instruction set**
- Important things to look at when comparing instruction sets
  - Word length
  - Verticality / horizontality
  - Supported operations
  - Addressing modes
  - Load and stores
  - Predicate execution
  - Address space
- We will have a look on the instruction set of a number of processors
- Equally important is to consider software support

Advanced RISC Machine (ARM)

- ARM is a RISC machine
- ARM is a comodity – ARM (the company) licenses its architecture to companies who manufacture the CPU as a stand-alone integrated circuit, or embed the CPU into larger systems
- **http://www.arm.com** – lots of information
- We first concentrate on the functionality of the instruction set
- There exists different implementations for the same instruction set, and this can have implications in the way we write (efficient) code
    - 3-stage pipeline (fetch, decode, execute)
    - 5-stage pipeline (fetch, decode, execute, buffer/data, write-back)

ARM processor organization



- The word is 32 bits long (four bytes)
- The address is 32 bits long
- An address refers to a byte, not a word (word 0 is at location 0, word 1 is at 4, word 2 is at 8, etc.)
- The PC is incremented by 4 in the absence of a branch

- ARM operations cannot be performed directly on memory locations. Thus, data operands must first be loaded into the CPU and then stored back to main memory to save the results (Load/Store architecture)

ARM processor organization (cont'd)

- 16 general-purpose registers (R0–R15). Except for R15, they are identical
- R15 is **program counter** – it should not be overwritten for use in data operations, but it can be used as an operand in computations
- Other basic register: **Current Program Status Register** (CPSR)
  - CPSR(31) = **N**egative bit is set when the result is negative in 2's-complement arithmetic
  - CPSR(30) = **Z**ero bit is set when every bit of the result is zero
  - CPSR(29) = **C**arry bit is set when there is a carry out of the operation
  - CPSR(28) = **O**verflow bit is set when an operation results in an overflow
- These bits can be used to check the result of an arithmetic operation

ARM data instructions

- **Arithmetic**

ADD	Add
ADC	Add with carry
SUB	Subtract
SBC	Subtract with carry
RSB	Reverse subtract
RSC	Reverse subtract with carry
MUL	Multiply
MLA	Multiply and accumulate

ARM data instructions (cont'd)

- **Logical**

AND	Bit-wise and
ORR	Bit-wise or
EOR	Bit-wise exclusive-or
BIC	Bit clear

- **Shift/rotate**

LSL	Logical shift left (zero fill)
LSR	Logical shift right (zero fill)
ASL	Arithmetic shift left
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right extended with C

ARM comparison, move, and branch instructions

- **Comparison**

CMP	Compare
CMN	Negated compare
TST	Bit-wise test
TEQ	Bit-wise negated test

- **Move**

MOV	Move
MVN	Move negated

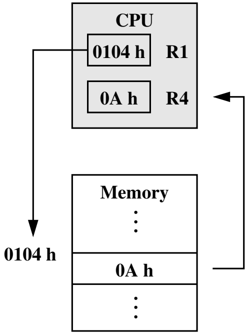
- **Branch**

B	Unconditional branch

ARM load-store instructions and pseudo-operations

- **Load-store**

LDR	Load
STR	Store
LDRH	Load half-word
STRH	Store half-word
LDRSH	Load half-word signed
LDRB	Load byte
STRB	Store byte



- **Pseudo**

ADR	Set register to address

```
ADR    R1, 0x104
LDR    R4, [R1]
```

ARM pseudo-operations

- `ADR` always assembles to one instruction
- A single `ADD` or `SUB` instruction is produced to load the address
- Example:

```
start       MOV   r0,#10
            ADR   r4,start    ; => SUB   r4,pc,#0xc
```

- Homework: check the other pseudo-operation: `ADRL`
- `ADRL` always assembles to two instructions

ARM uses 2's complement signed arithmetic

- 2's complement representation for a byte:

positive integer:	89	= 59 h	= **0101 1001**
negative integer:	-89	= a7 h	= **1010 0111**
positive integer:	23	= 17 h	= **0001 0111**
negative integer:	-12	= f4 h	= **1111 0100**

23 - 12 = 17 h - 0c h = 17 h + f4 h = 0b h = 11

- **Left-shifting** (multiplication by a power of 2) poses no problem
- Right-shifting ($\approx$ division by a power of 2) may be performed with or without sign extension depending on the number representation:

with sign extension:	$-89 >> 1$	$=$ a7 h $>> 1$	= d3 h =	-45
without sign extension:	$167 >> 1$	$=$ a7 h $>> 1$	= 53 h =	83

- Same problems may occur when loading and storing sub-word variables
- Consult the ARM databook and find out the semantics of the instructions that operate on signed and/or sub-word variables (shift/rotate, load/store)

ARM flow of control

- **Condition codes** (to be used as suffix for other instructions)

EQ	Equals zero	$Z = 1$
NE	Not equal to zero	$Z = 0$
CS	Carry set	$C = 1$
CC	Carry clear	$C = 0$
MI	Minus	$N = 1$
PL	Nonnegative (plus)	$N = 0$
VS	Overflow	$V = 1$
VC	No overflow	$V = 0$
HI	Unsigned higher	$C = 1$ and $Z = 0$
LS	Unsigned lower or same	$C = 0$ and $Z = 1$
GE	Signed greater than or equal	$N = V$
LT	Signed less than	$N \neq V$
GT	Signed greater than	$Z = 0$ and $N = V$
LE	Signed less than or equal	$Z = 1$ or $N \neq V$

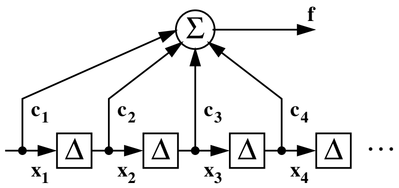
Compiler for ARM

- We shall use **gcc**, version 4.3.2 (as it is compatible with the ARM-Linux board connected to UVic-ECE network)
- It is installed in the ELW-B238 undergraduate lab
- You can login from remote
  $ ssh ugls.ece.uvic.ca
- It is available under /opt/arm/4.3.2/bin
- To generate the assembly file, use the "**−S**" flag
  $ /opt/arm/4.3.2/bin/arm-linux-gcc -S file.c
- Highly recommended: read the **gcc** documentation

Learning strategy

- It is of paramount importance to understand the translation from **C** to **assembly**
- We shall almost always use the "**-S**" flag
  $ /opt/arm/4.3.2/bin/arm-linux-gcc -S file.c
- How will the language constructs compile?
  - Assignments
  - Memory operations
  - Arithmetic operations
  - Logic operations
  - Loops (for, while)

ARM assembly code example – digital filter



```
i = 0;
f = 0;
while( i < N) {
  f = f + c(i) * x(i);
  i = i + 1;
}
```

```
MOV R0, #0   ; use R0 for i, set to 0
MOV R8, #0   ; use separate index
             ; for arrays
ADR R2, N    ; get address for N
LDR R1,[R2]  ; get the value of N for
             ; loop termination test
MOV R2, #0   ; use R2 for f, set to 0
ADR R3, c    ; load R3 with address
             ; of base of c array
ADR R5, x    ; load R5 with address
             ; of base of x array
```

```
loop LDR R4, [R3,R8] ; get value of c(i)
     LDR R6, [R5,R8] ; get value of x(i)
     MUL R4, R4, R6  ; compute c(i)*x(i)
     ADD R2, R2, R4  ; add into running
                     ; sum f
     ADD R8, R8, #4  ; add one word offset
                     ; to array index
     ADD R0, R0, #1  ; add 1 to i
     CMP R0, R1
     BLT loop          ; if i < N,
                       ; continue loop
```

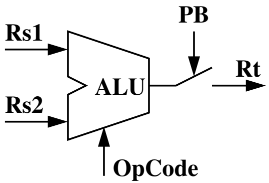
ARM programing example – `if` statement

- Let us assume a very simple C code

```
if( a < b)
  b = b - a;
else
  b = b + a;
```

- Let us write the assembly code implementing this code
- Two solutions are possible as we will see:
  - With extensive use of branches – classic way
  - With extensive use of predicates – there is no branch; instead, a predicate controls which instructions are executed
- Which one is better?

Predicate operations and architectural support



Instruction Format

OpCode	PB	Rs1	Rs2	Rt

Instruction format fields:

- **OpCode** – operation code
- **PB** – predicate (guard) bit
- **Rs1** – first source register
- **Rs2** – second source register
- **Rt** – target (result) register

Execution of predicate operations

- Operation Code is extracted, decoded, and issued
- Source operands are loaded
- Operation is performed
- Result is committed only if the predicate bit is set

Advantages and Disadvantages of Predicate Operations

A: It avoid jumps, increasing the effectiveness of pipelined execution

A: Certain bitwise operations may be quicker to evaluate

D: It increases the foorprint in the program memory

D: It complicates the hardware, thus it increases the critical path

<u>Pentium and ARM info</u>:

- Both x86-64 and 32-bit ARM support predication
- A number of predicates are available (depending on the flag register)

ARM programing example – `if` statement (cont’d)

The solution with branches

```
; compute and test the condition
   ADR R4, a     ; get address for a
   LDR R0, [R4] ; get value of a
   ADR R4, b     ; get address for b
   LDR R1, [R4] ; get value of b
   CMP R0, R1    ; compare a < b
   BGE fb        ; if a>=b, take branch
; a < b
   SUB R1, R1, R0 ; b = b - a
   B   j         ; go to joint
; a >= b
fb ADD R1, R1, R0 ; b = b + a
; save b
j  ADR R5, b     ; get address for b
   STR R1, [R5] ; store value of b
```

The solution with predicates

```
; compute and test the condition
   ADR R4, a        ; get address for a
   LDR R0, [R4]     ; get value of a
   ADR R4, b        ; get address for b
   LDR R1, [R4]     ; get value of b
   CMP R0, R1       ; compare a < b
; no branch here!
   SUBLT R1, R1, R0 ; b = b - a
; again no branch!
   ADDGE R1, R1, R0 ; b = b + a
; save b
   ADR R5, b        ; get address for b
   STR R1, [R5]     ; store value of b
```

Which solution is better?

- Hard to say – we may need to look at the implementation
- Let us assume a particular implementation of the ARM processor in which:
  - up to two instructions can be executed per cycle (2-issue slot ARM)
  - rule of thumb about latencies:
    - conditional branch takes at least 3 cycles to complete (one cycle to evaluate the condition expression, one cycle to analyze the outcome of the condition expression and decide whether to take the branch or not, one cycle to update the program counter)
    - multiplication takes 3 cycles to complete
    - all the other instructions (addition, subtraction, load, store, compare, ...) take 1 cycle to complete
  - there is a single computing unit for each operation
- Let's see the computing performance reported in number of cycles

Which solution is better (cont'd)

Solution with branches

```
   ADR R4, a        | NOP
   ADR R4, b        | LDR R0, [R4]
   NOP              | LDR R1, [R4]
   CMP R0, R1       | NOP
   BGE fb           | NOP
   NOP              | NOP
   NOP              | NOP
   SUB R1, R1, R0   | B   j
   NOP              | NOP
   NOP              | NOP
fb ADD R1, R1, R0   | NOP
j  ADR R5, b        | NOP
   STR R1, [R5]     | NOP
```

Solution with predicates

```
ADR R4, a          | NOP
ADR R4, b          | LDR R0, [R4]
NOP                | LDR R1, [R4]
CMP R0, R1         | NOP
SUBLT R1, R1, R0   | ADDGE R1, R1, R0
ADR R5, b          | NOP
STR R1, [R5]       | NOP
```

What you see is not ARM machine code – I used ARM assembly language just to describe the phenomena that would happen at implementation level when up to two instructions can be executed per cycle

- Predicates are good for relatively small conditionals
- When too many instructions fail the conditional test, the time required to evaluate those instructions outweighs other gains

Power Architecture and Processors (IBM)

- Both Power and ARM are RISC architectures
- Most Power internal registers are 64 bits wide
- ARM registers are general purpose and 32 bits wide
- Power has also dedicated registers (such as an Accumulator)
- Power is more powerful in signal processing
- Both ARM and Power architectures are open

- Power architecture is the major competitor to ARM

8051

- 8-bit CISC engine with register-memory architecture
- 64K program and data memory address space
- Support for 1-bit variables as a separate data type, allowing direct bit manipulation in control and logic systems that require Boolean processing
    - 17 dedicated instructions
    - Boolean accumulator
- Bit addressing is used especially for pin monitoring or program control flags
- Accumulator parity bit is available in the Program Status Word
- The is no Exclusive-OR operation – there are no common mechanical devices or relays analogous to Exclusive-OR operation
- Heterogenous instruction set – is this something to worry about?
- 80251 = 8051 augmented with operations on 16-bit and 32-bit words
- Free documentation: **http://www.intel.com**

68HC11

- CISC engine with register-memory architecture (one operand is the accumulator, the other is a memory location)
- Registers:
  - Two 8-bit accumulators A and B or a 16-bit accumulator D
  - Two 16-bit index registers IX and IY
  - 16-bit stack pointer register SP
  - 16-bit program counter register PC
  - 8-bit consition code register CCR: Carry/Borrow, Overflow, Zero, Negative, I-Interrupt Mask, Half Carry, X-Interrupt Mask, Stop Disable
- Simple instruction set
- Rich addressing capabilities, since the architecture is register-memory
- Standard 64 Kbyte addressing space
- A number of peripherals are integrated on chip

Emphasize on ARM

- Why ARM and not 8051/68HC11?
- 8051 is a microcontroller: extensive support for 1-bit variables, thus it is good for bit-level control (think of a wash-disher)
- 68HC11 belongs to the same class as 8051 – they are CISC machines
- We target the DSP domain which is computationally intensive
  - We need a RISC machine
  - We need a word width of 32 bits or more
- ARM is currently supported by GNU
- Linux is ported on ARM (32-bit ELF executable format)
- You can benefit from all existing software available in source format, thus ARM is a good platform for learning
- Try to install **arm-gcc** on your Intel-based workstation running Linux

Learning ARM

- To write good programs for ARM we need:
  - to know well the ARM architecture (that is, its instruction set)
  - to know well the limitations of the debugger and simulator
  - to understand what **the compiler can do for us** (that is, the translation process from high-level code to assembly)
- To learn the instruction set and the assembly language, you typically need to read the **Instruction Set Architecture** manual:
  - focus on what kind of operations are available (for example, fixed-point addition, or fixed-point fractional multipler)
  - do not spend too much time on the syntax of the assembly language, since you are going to program in high-level language anyway
- To learn the debugger and simulator you need to practice a lot
  - write simple programs to exercise the processor computation capabilities

Learning ARM (cont'd)

- To learn the compiler you need to:
  - Focus on the **Procedure Call Standard**, which is a collection of rules which governs calls between separately compiled or assembled code fragments:
    - constraints on the use of registers;
    - stack conventions;
    - argument passing and result return;
  - Understand the translation process from high-level language construct to machine-level instructions
    - A simple example: program for addition
    - A simple example: program for multiplication-by-constant
- Recommended literature
  - The GNU C compiler `http://gcc.gnu.org`
  - Writing efficient C for ARM, ARM Document No. DAI 0034A
  - ARM Developer Suite – Developer Guide, ARM Document No. DUI 0056D

ARM Procedure Call Standard

R15	Program Counter	PC	
R14	Link Register	LR	Address of the next instruction after BL instruction
R13	Stack Pointer	SP	Lower end of the stack frame
R12	Intra-Procedure	IP	Intra-Procedure-call scratch register
R11	Frame Pointer	FP	ARM-state variable register 8
R10			ARM-state variable register 7
R9			ARM-state variable register 6
R8			ARM-state variable register 5
R7			Variable register 4
R6			Variable register 3
R5			Variable register 2
R4			Variable register 1
R3			Argument/result/scratch register 4
R2			Argument/result/scratch register 3
R1			Argument/result/scratch register 2
R0			Argument/result/scratch register 1

Parameter passing and result return

- Parameter passing
  - For subroutines that take a small number of parameters, only registers are used (greatly reducing the overhead of a call)
  - The first 4 arguments are sent through registers R0 - R3
  - The next arguments are sent through the stack
- Result return
  - A fundamental data type that is smaller than 4 bytes is zero- and signed-extended and returned in R0
  - A word-sized fundamental data type (e.g., `int`, `float`) is returned in R0
  - A double-word sized fundamental data type (e.g., `long`, `double`) is returned in R0 and R1
  - A composite type not larger than 4 bytes is returned in R0
  - A composite type larger than 4 bytes is stored in memory as an address passed as an extra argument

A simple program for addition

```
#include <stdio.h>

int add( int a, int b) {
  return a + b;
}

int main( void) {
  int a = 1, b = 2, c;

  c = add( a, b);
  printf( "a + b = %i\n", c);
}
```

To compile just type: **arm-linux-gcc -static -S addition.c** <enter>

A simple program for addition

```
add:
        mov     ip, sp
        stmfd   sp!, {fp,ip,lr,pc}
        sub     fp, ip, #4
        sub     sp, sp, #8
        str     r0, [fp, #-16]
        str     r1, [fp, #-20]
        ldr     r3, [fp, #-16]
        ldr     r2, [fp, #-20]
        add     r3, r3, r2
        mov     r0, r3
        b       .L2
.L2:
        ldmea   fp, {fp,sp,pc}
```

- Contents of a processor registers are pushed onto the stack
- Passed parameters are copied onto the stack
- Function is called
- Function retrieves the data off the stack
- Function body is executed
- Return value is pushed onto the stack when the `return` is executed
- Calling function retrieves the return value off the stack
- Previous register contents are restored off the stack

A simple program for addition

```
add:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #8
        mov     r3, r0
        mov     r2, r1

        add     r3, r3, r2
        mov     r0, r3

        b       .L2
.L2:
        ldmea   fp, {fp, sp, pc}
```

A simple program for addition

```
add:
          mov      ip, sp
          stmfd    sp!, {fp, ip, lr, pc}
          sub      fp, ip, #4
          mov      r3, r0
          mov      r2, r1
          add      r3, r3, r2
          mov      r0, r3
          b        .L2
.L2:
          ldmea    fp, {fp, sp, pc}
```

To assemble just type: **arm-linux-gcc -static addition.s** <enter>

Reducing the function call overhead by inlining

- Inline functions: the body of the inline function is included into the calling function at compilation
- Methods to inline functions:
  - `inline` keyword

    ```
    inline int add( int a, int b) {
      ...
    }
    ```

  - Pragmas
  - Compiler options
- Guess what happened when I declared the function as `inline`?

Inline C code

```
add:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        mov     r3, r0
        mov     r2, r1
        add     r1, r3, r2
        mov     r0, r1
        b       .L2
.L2:
        ldmea   fp, {fp, sp, pc}
```

- The function call overhead reduced significantly but did not dissapear
- Use compiler optimization:

arm-linux-gcc -static -S -O3 addition.c <enter>

Inline C code

```
add:
        add     r0, r1, r0
        bx      lr

main:
        ...
        add     r3, r3, r1
        ...
        bl      printf
        ...
```

- Much simpler routine
- Inline routine

Inlining assembly code

- Basic inline assembly is not very convenient, since it is difficult to address machine registers from C code

```
int a, b, c;
__asm__ ( "add r0, r1, r2" );
```

- **gcc extended inline assembly**:

```
int a, b, c;
__asm__ ( "add %1, %2, %0" : "=r" (c) : "r" (a), "r" (b));
```

- Addition facilities may be provided, e.g., **inline assembler** (ARM specific)

```
int a, b, c;
__asm { add c, a, b; }
```

Using the Inline Assembler

- You must avoid using C variables with the same names as physical registers (e.g. r0, r1, etc.) or ARM Procedure Call Standard (APCS) named registers (e.g. ip, sl, etc.).
- If you try to access such a variable in an __asm block, the physical register will be accessed instead of the C variable, which may give unexpected results:

```
int fn( int v) {
  int ip;                   /* avoid C variable names like this! */
  __asm { add ip, v, #1; } /* the physical register ip is used
                              here, not the C variable ip */
  return ip;
}
```

Inline assembly code

C code:

```
int a, b, c;
a = 1;
b = 2;
__asm__ ( "add\t%0, %1, %2 : "=r" (c) : "r" (a), "r" (b));
```

Assembly code:

```
mov      r3, #1
str      r3, [fp, #-16]
mov      r3, #2
str      r3, [fp, #-20]
ldr      r3, [fp, #-16]
ldr      r2, [fp, #-20]
add      r3, r2, r3
mov      r2, r3
str      r2, [fp, #-24]
```

A simple program for multiplication

```
#include <stdio.h>

inline int mult( int a, int b) {
  return a * b;
}

int main( void) {
  int a = 1, b = 2, c;

  c = mult( a, b);
  printf( "a * b = %i\n", c);
}
```

To compile just type: **arm-linux-gcc -static -S multiplication.c** <enter>

A simple program for multiplication (cont'd)

Assembly code:

```
mult:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        mov     r3, r0
        mov     r2, r1
        mul     r1, r2, r3
        mov     r0, r1
        b       .L2
.L2:
        ldmea   fp, {fp, sp, pc}
```

- The general `MUL` instruction is used for multiplication
- What if only a multiplication-by-constant is needed?

A simple program for multiplication-by-constant

```
#include <stdio.h>

inline int mult_by_9( int a) {
  return a * 9;
}

int main( void) {
  int a = 1, c;

  c = mult_by_9( a);
  printf( "a * 9 = %i\n", c);
}
```

To compile just type: **arm-linux-gcc -static -S multiplication.c** <enter>

A simple program for multiplication-by-constant (cont'd)

Assembly code:

```
mult_by_9:
      mov     ip, sp
      stmfd   sp!, {fp, ip, lr, pc}
      sub     fp, ip, #4
      mov     r3, r0
      mov     r2, r3
      mov     r2, r2, asl #3
      add     r1, r2, r3
      mov     r0, r1
      b       .L3
.L3:
      ldmea   fp, {fp, sp, pc}
```

- The multiplication of `a` by constant `9` has been replaced by `(a « 3) + a`

Expose multiplication-by-constant algorithm to the compiler

```
#include <stdio.h>

inline int mult_by_9_exposed( int a) {
  return (a << 3) + a;
}

int main( void) {
  int a = 1, c;

  c = mult_by_9_exposed( a);
  printf( "a * 9 = %i\n", c);
}
```

To compile just type: **arm-linux-gcc -static -S multiplication.c** <enter>

Expose multiplication-by-constant algorithm to the compiler

Assembly code:

```
mult_by_9_exposed:
      mov     ip, sp
      stmfd   sp!, {fp, ip, lr, pc}
      sub     fp, ip, #4
      mov     r3, r0
      mov     r2, r3, asl #3
      add     r2, r2, r3
      mov     r0, r2
      b       .L4
.L4:
      ldmea   fp, {fp, sp, pc}
```

- The code is smaller with one instruction
- Check if an instruction like `add r0, r0, r0, asl #3` is possible

Carry and Overflow bits

- Very good tutorial:

  http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt
- Carry (CY) flag is used in unsigned arithmetic
- Overflow (OV) flag is used in signed arithmetic
- ALU sets these flags appropriately when doing integer arithmetic
- The ALU *doesn't know* about signed/unsigned
- **The programmer (that is, you!) needs to know which flag to check after the math is done**

Carry and Overflow bits

- **Unsigned arithmetic**
  - Watch the carry flag
  - Overflow flag bears no information
- **Signed arithmetic**
  - Watch the overflow flag
  - Carry flag bears no information
- **Carry (CY) flag** is set when the arithmetic operation causes a carry out of the most significant (32nd) position
- **Overflow (OV) flag** is set when both operands are of the same sign and the result is of opposite sign

Carry and Overflow bits – C code

```
#include <stdio.h>

int main( void) {
  unsigned int a, b;

  printf( "a = ");
  scanf( "%x", &a);
  printf( "b = ");
  scanf( "%x", &b);

  if( test_CY( a, b))
    printf( "CY set!\n");
  else
    printf( "CY not set!\n");

  if( test_OV( a, b))
    printf( "OV set!\n");
  else
    printf( "OV not set!\n");

  return (1);
}
```

Compile the code:
arm-linux-gcc -static source.c

Carry and Overflow bits – C code

```
unsigned int test_OV( unsigned int a, unsigned int b) {
  int OV;

  __asm__ __volatile__ (
    "mov\tr4, #0\n\t"                           mov      r4, #0
    "adds\tr3, %1, %2;\n\t"                     adds     r3, r0, r1;
    "movvs\tr4, #1\n\t"                         movvs    r4, #1
    "mov\t%0, r4\n\t"                           mov      r0, r4
    : "=r" (OV)
    : "r" (a), "r" (b)
    : "r3", "r4"
  );

  return OV;
}
```

Carry and Overflow bits – C code

```
unsigned int test_CY( unsigned int a, unsigned int b) {
  int CY;

  __asm__ __volatile__ (
    "mov\tr4, #0\n\t"                          mov     r4, #0
    "adds\tr3, %1, %2;\n\t"                    adds    r3, r0, r1;
    "movcs\tr4, #1\n\t"                        movcs   r4, #1
    "mov\t%0, r4\n\t"                          mov     r0, r4
    : "=r" (CY)
    : "r" (a), "r" (b)
    : "r3", "r4"
  );

  return CY;
}
```

Carry and Overflow bits – Executable

```
[user1@FriendlyARM]# ./a.out
a = ffff
b = ffff
CY not set!
OV not set!
```

```
[user1@FriendlyARM]# ./a.out
a = 7fffffff
b = 7fffffff
CY not set!
OV set!
```

```
[user1@FriendlyARM]# ./a.out
a = ffffffff
b = ffffffff
CY set!
OV not set!
```

```
[user1@FriendlyARM]# ./a.out
a = ffffffff
b = 80000000
CY set!
OV set!
```

ARM Architecture – Review

- **ARM** instruction set
- **Thumb** is a 16-bit encoding for a subset of the ARM instruction set
    - Increased code density is achieved due to implicit instruction operands
    - Only half of the registers can be accessed
    - Only branches can be conditional
- **Vector Floating Point (VFP)** instruction set
    - Single/double-precision floating-point computation
    - This mode was replaced by NEON mode
- **Advanced SIMD (NEON)** is a 64/128-bit instruction set
    - Supports vectors of integers and floating-point values
    - NEON and VFP units share the same floating-point registers

NEON SIMD Intrinsics – C Code

- The `arm_neon.h` header needs to be included

```
#include "arm_neon.h"

uint32x2_t aa, bb, ss;

int main( void) {
  ss = vadd_u32( aa, bb);
}
```

- To compile:
  $ arm-linux-gcc -mfloat-abi=softfp -mfpu=neon -static -O3 -S file.c

NEON SIMD Intrinsics – Assembly

```
main:
        ldr         r3, .L3
        ldr         r2, .L3+4
        fldd        d17, [r3, #0]
        fldd        d16, [r2, #0]
        ldr         r3, .L3+8
        vadd.i32    d17, d17, d16
        fstd        d17, [r3, #0]
        bx          lr
```

Questions, feedback

Notes I

Notes II

Notes III

Notes IV