

# Engineering Cryptographic Software

An introduction to the Cortex-M4

Radboud University, Nijmegen, The Netherlands



Winter 2022

## Our platform: Arm

- ▶ Company designs CPUs, does not build them
- ▶ Market leader for mobile devices, embedded systems
  
- ▶ ARMv7E-M architecture
- ▶ Cortex-M4 implements this architecture
- ▶ Released in 2010, widely deployed

# Our platform: Arm

- ▶ Company designs CPUs, does not build them
- ▶ Market leader for mobile devices, embedded systems
- ▶ ARMv7E-M architecture
- ▶ Cortex-M4 implements this architecture
- ▶ Released in 2010, widely deployed
- ▶ STM32F407VGT6
  - ▶ Cortex-M4 + peripherals
- ▶ 1024 KB flash
- ▶ 192 KB SRAM
- ▶ 168 MHz CPU

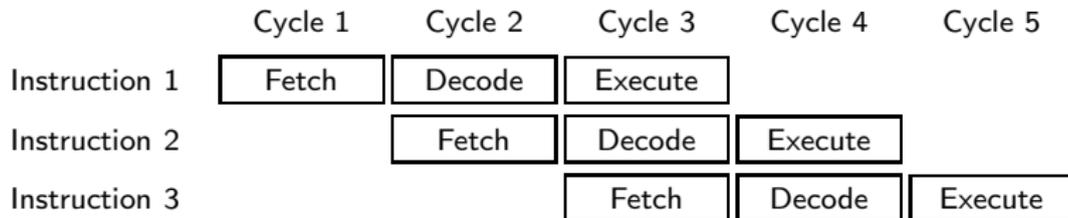


# Pipeline

- ▶ Cortex-M4 has pipelined execution

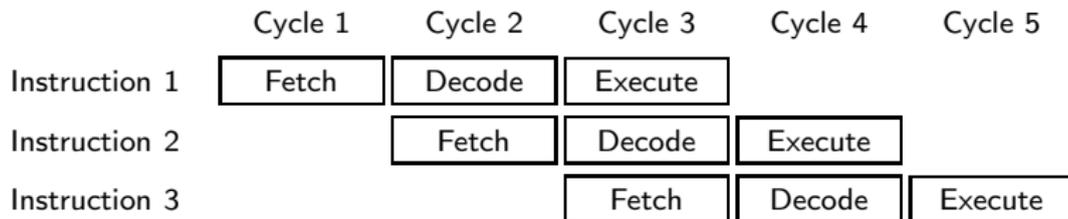
# Pipeline

- ▶ Cortex-M4 has pipelined execution
- ▶ 3 stages: fetch, decode, execute



# Pipeline

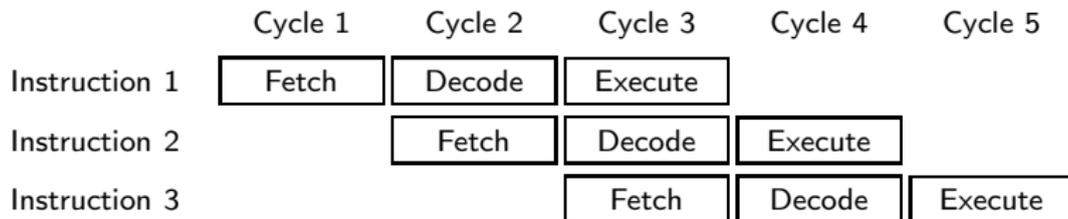
- ▶ Cortex-M4 has pipelined execution
- ▶ 3 stages: fetch, decode, execute



- ▶ Branching breaks this
  - ▶ But remedied by branch prediction + speculative execution

# Pipeline

- ▶ Cortex-M4 has pipelined execution
- ▶ 3 stages: fetch, decode, execute



- ▶ Branching breaks this
  - ▶ But remedied by branch prediction + speculative execution
- ▶ Execute happens in one cycle: dependencies do not cause stalls

# Caches

- ▶ Access to RAM on the Cortex-M4 by itself is not cached

# Caches

- ▶ Access to RAM on the Cortex-M4 by itself is not cached
- ▶ STM32F407 has cache to flash memory
- ▶ Lookups from constant tables go through cache → **timing leakage!**

# Caches

- ▶ Access to RAM on the Cortex-M4 by itself is not cached
- ▶ STM32F407 has cache to flash memory
- ▶ Lookups from constant tables go through cache → **timing leakage!**
- ▶ Binaries also run on Cortex-M7, which has cached access to RAM
  
- ▶ Write “constant-time” code!
  - ▶ No branching on secret data
  - ▶ No memory access at secret locations

# Registers

- ▶ 16 registers: r0–r15

# Registers

- ▶ 16 registers: r0–r15
- ▶ Some special registers
  - ▶ r13: sp (stack pointer)
  - ▶ r14: lr (link register)
  - ▶ r15: pc (program counter)

# Registers

- ▶ 16 registers: r0–r15
- ▶ Some special registers
  - ▶ r13: sp (stack pointer)
  - ▶ r14: lr (link register)
  - ▶ r15: pc (program counter)
- ▶ r0–r12 are general purpose and can be freely used
- ▶ r14 can be used inside a function if spilled and restored before return
- ▶ r13 and r15 should be used only for their purpose

# Instructions

- ▶ Format: Instr Rd, Rn(, Rm)

# Instructions

- ▶ Format: `Instr Rd, Rn(, Rm)`
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)

# Instructions

- ▶ Format: `Instr Rd, Rn(, Rm)`
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`

# Instructions

- ▶ Format: Instr Rd, Rn(, Rm)
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`
  - ▶ Sometimes, a constant is too large to fit in an instruction
  - ▶ Put constant in memory (see later) or construct it
  - ▶ `movw` for bottom 16 bits, `movt` for top 16 bits

# Instructions

- ▶ Format: Instr Rd, Rn(, Rm)
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`
  - ▶ Sometimes, a constant is too large to fit in an instruction
  - ▶ Put constant in memory (see later) or construct it
  - ▶ `movw` for bottom 16 bits, `movt` for top 16 bits
- ▶ `add`, but also `adds`, `adc`, and `adcs`

# Instructions

- ▶ Format: Instr Rd, Rn(, Rm)
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`
  - ▶ Sometimes, a constant is too large to fit in an instruction
  - ▶ Put constant in memory (see later) or construct it
  - ▶ `movw` for bottom 16 bits, `movt` for top 16 bits
- ▶ `add`, but also `adds`, `adc`, and `adcs`
  - ▶ By default, flags never get updated!
  - ▶ Many instructions have a variant that sets flags by appending `s`

# Instructions

- ▶ Format: `Instr Rd, Rn(, Rm)`
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`
  - ▶ Sometimes, a constant is too large to fit in an instruction
  - ▶ Put constant in memory (see later) or construct it
  - ▶ `movw` for bottom 16 bits, `movt` for top 16 bits
- ▶ `add`, but also `adds`, `adc`, and `adcs`
  - ▶ By default, flags never get updated!
  - ▶ Many instructions have a variant that sets flags by appending `s`
- ▶ Bitwise operations: `eor`, `and`, `orr`, `mvn`

# Instructions

- ▶ Format: `Instr Rd, Rn(, Rm)`
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`
  - ▶ Sometimes, a constant is too large to fit in an instruction
  - ▶ Put constant in memory (see later) or construct it
  - ▶ `movw` for bottom 16 bits, `movt` for top 16 bits
- ▶ `add`, but also `adds`, `adc`, and `adcs`
  - ▶ By default, flags never get updated!
  - ▶ Many instructions have a variant that sets flags by appending `s`
- ▶ Bitwise operations: `eor`, `and`, `orr`, `mvn`
- ▶ Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`

# Instructions

- ▶ Format: Instr Rd, Rn(, Rm)
- ▶ `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- ▶ `mov r0, #18`
  - ▶ Sometimes, a constant is too large to fit in an instruction
  - ▶ Put constant in memory (see later) or construct it
  - ▶ `movw` for bottom 16 bits, `movt` for top 16 bits
- ▶ `add`, but also `adds`, `adc`, and `adcs`
  - ▶ By default, flags never get updated!
  - ▶ Many instructions have a variant that sets flags by appending `s`
- ▶ Bitwise operations: `eor`, `and`, `orr`, `mvn`
- ▶ Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- ▶ All have variants with registers as operands and with a constant ('immediate')

## Combined barrel shifter

- ▶ Distinctive feature of Arm architecture
- ▶ Every  $R_m$  operand goes through barrel shifter
- ▶ Possible to do this: `eor r0, r1, r2, lsl #2`

## Combined barrel shifter

- ▶ Distinctive feature of Arm architecture
- ▶ Every  $R_m$  operand goes through barrel shifter
- ▶ Possible to do this: `eor r0, r1, r2, lsl #2`
- ▶ Two instructions for the price of one, only costs 1 cycle

## Combined barrel shifter

- ▶ Distinctive feature of Arm architecture
- ▶ Every  $R_m$  operand goes through barrel shifter
- ▶ Possible to do this: `eor r0, r1, r2, lsl #2`
- ▶ Two instructions for the price of one, only costs 1 cycle
- ▶ Optimized code uses this all the time
- ▶ Possible with most arithmetic instructions

## Barrel shifter example

Possible:

```
mov r0, #42
mov r1, #37
ror r1, r1, #1
orr r2, r0, r1
lsl r2, r2, #1
eor r0, r2
```

## Barrel shifter example

Possible:

```
mov r0, #42
mov r1, #37
ror r1, r1, #1
orr r2, r0, r1
lsl r2, r2, #1
eor r0, r2
```

More efficient:

```
mov r0, #42
mov r1, #37
orr r2, r0, r1, ror #1
eor r0, r0, r2, lsl #1
```

## Barrel shifter example

Possible:

```
mov r0, #42
mov r1, #37
ror r1, r1, #1
orr r2, r0, r1
lsl r2, r2, #1
eor r0, r2
```

More efficient:

```
mov r0, #42
mov r1, #37
orr r2, r0, r1, ror #1
eor r0, r0, r2, lsl #1
```

- ▶ Barrel shifter does not update Rm, i.e. r1 and r2!

## Branching and labels

- ▶ After every 32-bit instruction,  $pc += 4$
- ▶ By writing to the  $pc$ , we can jump to arbitrary locations (and continue execution from there)

## Branching and labels

- ▶ After every 32-bit instruction,  $pc += 4$
- ▶ By writing to the  $pc$ , we can jump to arbitrary locations (and continue execution from there)
- ▶ While programming, addresses of instructions are not known

## Branching and labels

- ▶ After every 32-bit instruction,  $pc += 4$
- ▶ By writing to the  $pc$ , we can jump to arbitrary locations (and continue execution from there)
- ▶ While programming, addresses of instructions are not known
- ▶ Solution: define a *label* and use  $b$  to branch to labels

## Branching and labels

- ▶ After every 32-bit instruction,  $pc += 4$
- ▶ By writing to the  $pc$ , we can jump to arbitrary locations (and continue execution from there)
- ▶ While programming, addresses of instructions are not known
- ▶ Solution: define a *label* and use  $b$  to branch to labels
- ▶ Assembler and linker later resolve the address

## Branching and labels

- ▶ After every 32-bit instruction,  $pc += 4$
- ▶ By writing to the  $pc$ , we can jump to arbitrary locations (and continue execution from there)
- ▶ While programming, addresses of instructions are not known
- ▶ Solution: define a *label* and use `b` to branch to labels
- ▶ Assembler and linker later resolve the address

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

## Conditional branches

- ▶ How to do a `for/while` loop?

## Conditional branches

- ▶ How to do a `for/while` loop?
- ▶ Need to do a *test* and branch depending on the outcome

## Conditional branches

- ▶ How to do a `for/while` loop?
- ▶ Need to do a *test* and branch depending on the outcome
  - ▶ `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - ▶ `cmp r0, #5`

## Conditional branches

- ▶ How to do a `for/while` loop?
- ▶ Need to do a *test* and branch depending on the outcome
  - ▶ `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - ▶ `cmp r0, #5`
- ▶ Really: subtract, set status flags, discard result

## Conditional branches

- ▶ How to do a `for/while` loop?
- ▶ Need to do a *test* and branch depending on the outcome
  - ▶ `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - ▶ `cmp r0, #5`
- ▶ Really: subtract, set status flags, discard result
- ▶ Instead of `b`, use a conditional branch
  - ▶ `beq label` (`r0 == r1`)
  - ▶ `bne label` (`r0 != r1`)

## Conditional branches

- ▶ How to do a `for/while` loop?
- ▶ Need to do a *test* and branch depending on the outcome
  - ▶ `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - ▶ `cmp r0, #5`
- ▶ Really: subtract, set status flags, discard result
- ▶ Instead of `b`, use a conditional branch
  - ▶ `beq label` (`r0 == r1`)
  - ▶ `bne label` (`r0 != r1`)
  - ▶ `bhi label` (`r0 > r1`, unsigned)
  - ▶ `bls label` (`r0 <= r1`, unsigned)
  - ▶ `bgt label` (`r0 > r1`, signed)
  - ▶ `bge label` (`r0 >= r1`, signed)

## Conditional branches

- ▶ How to do a `for/while` loop?
- ▶ Need to do a *test* and branch depending on the outcome
  - ▶ `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - ▶ `cmp r0, #5`
- ▶ Really: subtract, set status flags, discard result
- ▶ Instead of `b`, use a conditional branch
  - ▶ `beq label` (`r0 == r1`)
  - ▶ `bne label` (`r0 != r1`)
  - ▶ `bhi label` (`r0 > r1`, unsigned)
  - ▶ `bls label` (`r0 <= r1`, unsigned)
  - ▶ `bgt label` (`r0 > r1`, signed)
  - ▶ `bge label` (`r0 >= r1`, signed)
  - ▶ And many more

## Conditional branches (example)

► In C:

```
uint32_t a, b = 100;

for (a = 0; a <= 50; a++) {
    b += a;
}
```

► In asm:

```
mov r0, #0    // a
mov r1, #100  // b

loop:
add r1, r0    // b += a

add r0, #1    // a++
cmp r0, #50   // compare a and 50
bls loop     // loop if <=
```

# The stack

- ▶ Often data does not fit in registers

# The stack

- ▶ Often data does not fit in registers
- ▶ Solution: push intermediate values to the stack (changes `sp`)

# The stack

- ▶ Often data does not fit in registers
- ▶ Solution: push intermediate values to the stack (changes `sp`)
- ▶ `push {r0, r1}`

# The stack

- ▶ Often data does not fit in registers
- ▶ Solution: push intermediate values to the stack (changes `sp`)
- ▶ `push {r0, r1}`
- ▶ Can now re-use `r0` and `r1`

# The stack

- ▶ Often data does not fit in registers
- ▶ Solution: push intermediate values to the stack (changes sp)
- ▶ `push {r0, r1}`
- ▶ Can now re-use r0 and r1
- ▶ Later retrieve values in any register you like: `pop {r0, r2}`

# The stack

- ▶ Often data does not fit in registers
- ▶ Solution: push intermediate values to the stack (changes sp)
- ▶ `push {r0, r1}`
- ▶ Can now re-use r0 and r1
- ▶ Later retrieve values in any register you like: `pop {r0, r2}`
- ▶ Can load from the stack without moving sp (in a few slides)
- ▶ Not popping all pushed values will crash the program

# Memory

- ▶ Stack is nice for intermediate values, but not for constants or lookup tables

# Memory

- ▶ Stack is nice for intermediate values, but not for constants or lookup tables
- ▶ '*word*' = 32 bit, '*halfword*' = 16 bit, '*doubleword*' = 64 bit, '*byte*' = 8 bit, '*nibble*' = 4 bit

# Memory

- ▶ Stack is nice for intermediate values, but not for constants or lookup tables
- ▶ 'word' = 32 bit, 'halfword' = 16 bit, 'doubleword' = 64 bit, 'byte' = 8 bit, 'nibble' = 4 bit
- ▶ Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

# Memory

- ▶ Stack is nice for intermediate values, but not for constants or lookup tables
- ▶ 'word' = 32 bit, 'halfword' = 16 bit, 'doubleword' = 64 bit, 'byte' = 8 bit, 'nibble' = 4 bit
- ▶ Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- ▶ Ends up *somewhere* in RAM, need a label to access it

# Memory

- ▶ Stack is nice for intermediate values, but not for constants or lookup tables
- ▶ 'word' = 32 bit, 'halfword' = 16 bit, 'doubleword' = 64 bit, 'byte' = 8 bit, 'nibble' = 4 bit
- ▶ Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- ▶ Ends up *somewhere* in RAM, need a label to access it
- ▶ For  $n$  bytes of uninitialized memory, use a label and `.skip n`
  - ▶ For  $n$  bytes of 0-initialized data, use `.lcomm somelabel, n`

# Memory

- ▶ Stack is nice for intermediate values, but not for constants or lookup tables
- ▶ 'word' = 32 bit, 'halfword' = 16 bit, 'doubleword' = 64 bit, 'byte' = 8 bit, 'nibble' = 4 bit
- ▶ Can directly insert words and bytes as 'data'

```
.data
somedata:
    .word 0x01234567, 0xfedcba98
    .byte 0x2a, 0x25
.text
    //continue with code
```

- ▶ Ends up *somewhere* in RAM, need a label to access it
- ▶ For  $n$  bytes of uninitialized memory, use a label and `.skip n`
  - ▶ For  $n$  bytes of 0-initialized data, use `.lcomm somelabel, n`
- ▶ For global constants in ROM/flash, use `.section .rodata`

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value
- ▶ `ldr r1, [r0, #4]` loads from `r0+4` (bytes)

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value
- ▶ `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- ▶ `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- ▶ `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value
- ▶ `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- ▶ `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- ▶ `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- ▶ `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- ▶ `ldr r1, [r0, r2, lsl #2]` is possible
  - ▶ if `r2` was a byte-offset, it's now used as word-offset

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value
- ▶ `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- ▶ `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- ▶ `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- ▶ `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- ▶ `ldr r1, [r0, r2, lsl #2]` is possible
  - ▶ if `r2` was a byte-offset, it's now used as word-offset
- ▶ `str` also has these variants

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value
- ▶ `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- ▶ `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- ▶ `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- ▶ `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- ▶ `ldr r1, [r0, r2, lsl #2]` is possible
  - ▶ if `r2` was a byte-offset, it's now used as word-offset
- ▶ `str` also has these variants
- ▶ `ldm/stm r0, {r1,r2,r5}` loads/stores multiple from consecutive memory locations
- ▶ `ldm/stm r0!, {r1,r2,r5} [..]` and increments `r0`

## Using memory

- ▶ `adr r0, somelabel` to get the address in a register
- ▶ `ldr/str r1, [r0]` loads/stores a value
- ▶ `ldr r1, [r0, #4]` loads from `r0+4` (bytes)
- ▶ `ldr r1, [r0, #4]!` loads from `r0+4` and increments `r0` by 4
- ▶ `ldr r1, [r0], #4` loads from `r0` and increments `r0` by 4
- ▶ `ldr r1, [r0, r2]` loads from `r0+r2`, cannot increment
- ▶ `ldr r1, [r0, r2, lsl #2]` is possible
  - ▶ if `r2` was a byte-offset, it's now used as word-offset
- ▶ `str` also has these variants
- ▶ `ldm/stm r0, {r1,r2,r5}` loads/stores multiple from consecutive memory locations
- ▶ `ldm/stm r0!, {r1,r2,r5} [..]` and increments `r0`
- ▶ `push {r0,r1} == stmdb sp!, {r0,r1}`
  - ▶ *'store multiple decrement before'*

## Subroutines

```
somelabel:  
    add r0, r1  
    add r0, r1, ror #2  
    add r0, r1, ror #4  
    bx lr
```

```
main:  
    bl somelabel  
    mov r4, r0  
    mov r0, r2  
    mov r1, r3  
    bl somelabel
```

- ▶ lr keeps track of 'return address'
- ▶ Branch with link (bl) automatically sets lr

## Subroutines

```
somelabel:  
    add r0, r1  
    add r0, r1, ror #2  
    add r0, r1, ror #4  
    bx lr
```

```
main:  
    bl somelabel  
    mov r4, r0  
    mov r0, r2  
    mov r1, r3  
    bl somelabel
```

- ▶ lr keeps track of 'return address'
- ▶ Branch with link (bl) automatically sets lr
- ▶ Some performance overhead due to branching

# Application Binary Interface (ABI)

- ▶ Agreement on how to deal with parameters and return values

# Application Binary Interface (ABI)

- ▶ Agreement on how to deal with parameters and return values
- ▶ If it fits, parameters in `r0-r3`

# Application Binary Interface (ABI)

- ▶ Agreement on how to deal with parameters and return values
- ▶ If it fits, parameters in `r0-r3`
- ▶ Otherwise, a part in `r0-r3` and the rest on the stack

# Application Binary Interface (ABI)

- ▶ Agreement on how to deal with parameters and return values
- ▶ If it fits, parameters in `r0-r3`
- ▶ Otherwise, a part in `r0-r3` and the rest on the stack
- ▶ Return value in `r0`

# Application Binary Interface (ABI)

- ▶ Agreement on how to deal with parameters and return values
- ▶ If it fits, parameters in `r0-r3`
- ▶ Otherwise, a part in `r0-r3` and the rest on the stack
- ▶ Return value in `r0`
  
- ▶ The callee(!) should preserve `r4-r11` if it overwrites them
- ▶ `r12` is a scratch register (no need to preserve)
- ▶ Important when calling your assembly from, e.g., C

# Application Binary Interface (ABI)

- ▶ Agreement on how to deal with parameters and return values
- ▶ If it fits, parameters in `r0-r3`
- ▶ Otherwise, a part in `r0-r3` and the rest on the stack
- ▶ Return value in `r0`
  
- ▶ The callee(!) should preserve `r4-r11` if it overwrites them
- ▶ `r12` is a scratch register (no need to preserve)
- ▶ Important when calling your assembly from, e.g., C
  
- ▶ For *private* subroutines: can ignore this ABI

# Architecture Reference Manual

- ▶ Large PDF that includes all of this, and more
- ▶ Available online: <https://developer.arm.com/documentation/ddi0403/eb/>
- ▶ See Chapter A7 for instruction listings and descriptions

# Architecture Reference Manual

## A6.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

**Encoding T1** All versions of the Thumb ISA.

ADDS <Rd>, <Rn>, #<imm3>

Outside IT block.

ADD<C> <Rd>, <Rn>, #<imm3>

Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);

**Encoding T2** All versions of the Thumb ISA.

ADDS <Rdn>, #<imm8>

Outside IT block.

ADD<C> <Rdn>, #<imm8>

Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

**Encoding T3** ARMv7-M

ADD{S}<C> .W <Rd>, <Rn>, #<const>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
...																Rn		...															

# Architecture Reference Manual

## Assembler syntax

ADD{S}<C><Q> {<Rd>}, <Rn>, #<const>

ADDW<C><Q> {<Rd>}, <Rn>, #<const>

where:

S	If present, specifies that the instruction updates the flags. Otl update the flags.
<C><Q>	See <i>Standard assembler syntax fields</i> on page A6-7.
<Rd>	Specifies the destination register. If <Rd> is omitted, this reg
<Rn>	Specifies the register that contains the first operand. If the S  ( <i>SP plus immediate</i> ) on page A6-26. If the PC is specified fo
<const>	Specifies the immediate value to be added to the value obta allowed values is 0-7 for encoding T1, 0-255 for encoding T See <i>Modified immediate constants in Thumb instructions</i> o allowed values for encoding T3.

# Time to get to work!

- ▶ If you haven't "walked through" the STM32F4 getting started, do so.
- ▶ If you never wrote assembly before, download this example in your assignment folder:  
[https://github.com/denigreco/  
crypto\\_engineering\\_asm\\_example.git](https://github.com/denigreco/crypto_engineering_asm_example.git)
- ▶ Otherwise, start working on ChaCha20
- ▶ These slides are also on the course website