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

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td>Instruction 2</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td>Instruction 3</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
</tr>
</tbody>
</table>
Pipeline

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

<table>
<thead>
<tr>
<th></th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 2</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 3</td>
<td>Fetch</td>
<td>Decode</td>
<td></td>
<td>Execute</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction 2</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction 3</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Branching breaks this
  - But remedied by branch prediction + speculative execution
- Execute happens in one cycle: dependencies do not cause stalls
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!**
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
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 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 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 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 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 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 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 Rm operand goes through barrel shifter
- Possible to do this: eor r0, r1, r2, lsl #2
Combined barrel shifter

- Distinctive feature of ARM architecture
- Every Rm 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 Rm 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:

\[
\begin{align*}
\text{mov} & \ r0, \ #42 \\
\text{mov} & \ r1, \ #37 \\
\text{ror} & \ r1, \ r1, \ #1 \\
\text{orr} & \ r2, \ r0, \ r1 \\
\text{lsl} & \ r2, \ r2, \ #1 \\
\text{eor} & \ r0, \ r2
\end{align*}
\]

More efficient:

\[
\begin{align*}
\text{mov} & \ r0, \ #42 \\
\text{mov} & \ r1, \ #37 \\
\text{orr} & \ r2, \ r0, \ r1, \ \text{ror} \ #1 \\
\text{eor} & \ r0, \ r0, \ r2, \ \text{lsl} \ #1
\end{align*}
\]

▸ 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, \( \text{pc} += 4 \)
- By writing to the \( \text{pc} \), 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, \( \text{pc} += 4 \)
- By writing to the \( \text{pc} \), we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a \textit{label} and use \texttt{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:

```c
uint32_t a, b = 100;

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

- In asm:

```assembly
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
- Not popping all pushed values will crash the program
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 the
- 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 the
- 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
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsigned multiplication yielding 64-bit result)
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - **UMULL** (long unsigned multiplication yielding 64-bit result)
- **DSP/SIMD** instructions operate on multiple values at once
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsigned multiplication yielding 64-bit result)
- DSP/SIMD instructions operate on multiple values at once
  - UADD16 (16-bit addition on two values packed in one register)
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsinged multiplication yielding 64-bit result)
- DSP/SIMD instructions operate on multiple values at once
  - UADD16 (16-bit addition on two values packed in one register)
  - SMUAD (multiplication of the half-words; sum of the two products)
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsigned multiplication yielding 64-bit result)
- DSP/SIMD instructions operate on multiple values at once
  - UADD16 (16-bit addition on two values packed in one register)
  - SMUAD (multiplication of the half-words; sum of the two products)
  - ...
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsigned multiplication yielding 64-bit result)
- DSP/SIMD instructions operate on multiple values at once
  - UADD16 (16-bit addition on two values packed in one register)
  - SMUAD (multiplication of the half-words; sum of the two products)
  - ...
- We are working on M4F, i.e., we have a floating point unit
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsigned multiplication yielding 64-bit result)
- DSP/SIMD instructions operate on multiple values at once
  - UADD16 (16-bit addition on two values packed in one register)
  - SMUAD (multiplication of the half-words; sum of the two products)
  - ...
- We are working on M4F, i.e., we have a floating point unit
  - For crypto, the arithmetic is usually not that useful
More advanced topics

- The M4 has powerful 1-cycle multiplication instructions
  - UMULL (long unsigned multiplication yielding 64-bit result)
- DSP/SIMD instructions operate on multiple values at once
  - UADD16 (16-bit addition on two values packed in one register)
  - SMUAD (multiplication of the half-words; sum of the two products)
- ... 
- We are working on M4F, i.e., we have a floating point unit
  - For crypto, the arithmetic is usually not that useful
  - Can for example use for register spills, as a loop counter, ...

Architecture Reference Manual

- Large PDF that includes all of this, and more
- Available online: https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf
- See Chapter A7 for instruction listings and descriptions
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>
ADD<s> <Rd>,<Rn>,#<imm3>

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0 imm3 Rn Rd</td>
</tr>
</tbody>
</table>

\[ d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn}); \quad \text{setflags} = \neg \text{ITBlock}(); \quad \text{imm32} = \text{ZeroExtend}(\text{imm3}, 32); \]

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

ADDs <Rdn>,#<imm8>
ADD<s> <Rdn>,#<imm8>

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 Rdn imm8</td>
</tr>
</tbody>
</table>

\[ d = \text{UInt}(\text{Rdn}); \quad n = \text{UInt}(\text{Rdn}); \quad \text{setflags} = \neg \text{ITBlock}(); \quad \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32); \]

**Encoding T3**  ARMv7-M

ADD{s}<c>.W <Rd>,<Rn>,#<const>

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
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. Otherwise
update the flags.


<Rd> Specifies the destination register. If <Rd> is omitted, this reg

<Sp> Specifies the register that contains the first operand. If the S
(SP plus immediate) on page A6-26. If the PC is specified for

<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 Modified immediate constants in Thumb instructions or
allowed values for encoding T3.
Getting Started

- We are working on two different boards
  - STM32F407 with 192 KiB of RAM
  - STM32F413 with 320 KiB of RAM
- Writing code for each of them is exactly the same, but the setup is slightly different
- Follow steps on https://github.com/mkannwischer/STM32-getting-started
  - Install arm-none-eabi-gcc
  - Install st-link
STM32F407: Connecting discovery board

- Connect USB cable to your machine
  - Used for flashing and as power supply
- Connect PA2 pin with RXD pin of UART-USB connector
  - Used for receiving serial output
  - You may also connect GND with GND
STM32F407: Flashing some examples

- Clone getting started
  ```
git clone --recurse-submodules https://github.com/mkannwischer/STM32-getting-started
  ```

- Compile libopencm3 library
  ```
cd STM32-getting-started/libopencm3
make
```

- Compile binaries for your board
  ```
cd STM32F407
make
```

- Flash binary to the board
  ```
st-flash write usart.bin 0x8000000
```

- Receive output
  ```
screen /dev/ttyUSB0 115200
or
`./host_unidirectional.py`
```
STM32F413

- One USB cable for power, flashing, and receiving USART output
- Clone getting started
  
git clone --recurse-submodules
  https://github.com/mkannwischer/STM32-getting-started

- Compile libopencm3 library
  
  cd STM32-getting-started/libopencm3
  
  make
  
  cd ..

- Compile binaries for your board
  
  cd STM32F413
  
  make

- Flash binary to the board
  
  st-flash write usart.bin 0x8000000

- Receive output
  
  screen /dev/ttyACMO 115200
Once familiar with the board, you may want to try some crypto

PQM4 is testing and benchmarking frame work for post-quantum crypto

- [https://github.com/mupq/pqm4](https://github.com/mupq/pqm4)
- Contains a collection of post-quantum signatures and KEMs
- Some are assembly optimized
- Primarily targetting the STM32F407

Getting started

```shell
git clone --recurse-submodules git@github.com:mupq/pqm4
cd pqm4
./build_everything.py kyber768
st-flash write bin/crypto_kem_kyber768_m4_test.bin 0x8000000
```

For the STM32F413 there is a separate branch:

[https://github.com/mupq/pqm4/tree/f413](https://github.com/mupq/pqm4/tree/f413)