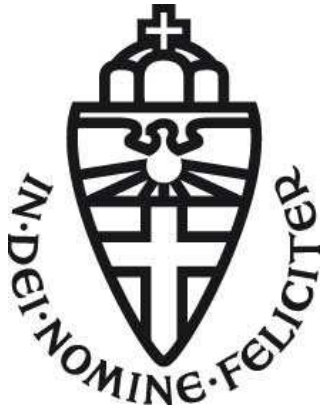
Cryptographic Engineering

Cryptography in software – the basics

Radboud University, Nijmegen, The Netherlands



Spring 2019

The software arena(s)

Embedded microcontrollers

- This is what you're looking at in the software assignment
- Typically very tight size constraints (ROM and RAM)
- Different optimization targets: size, speed
- No (or very little) parallel computation capabilities

Servers, workstations, laptops, smartphones

- No *serious* size constraints for crypto
- Optimization target: speed (high throughput or **low latency**)
- Various different levels of parallelism

GPUs

- Special size restrictions apply for good performance
- Optimization target: speed (**high throughput** or low latency)
- Highly parallel architectures

Throughput vs. Latency

- Some software makes extensive use of *batching*
- Faster for many computations, if those are performed "together"
- Example: McBits software (Bernstein, Chou, Schwabe, 2013):
  - 15486208 cycles on Intel Ivy Bridge for 256 decryptions
  - **NOT:** $15486208/256 = 60493$ cycles for one decryption.
  - Software needs to wait until enough inputs are available
  - Delay from input to output is delay of 256 decryptions
- Highly parallel architectures (e.g., GPUs) focus on throughput
- This can be a problem for, e.g., low-latency network communication

Benchmarking software

- Tools like `time` or `time.h` have too low resolution
- For serious optimization need to count CPU cycles
- Use CPU's built-in cycle counter, e.g., on AMD64:

```
static long long cpucycles(void)
{
  unsigned long long result;
  asm volatile("rdtsc;"
               "shlq $32,%%rdx;"
               "orq %%rdx,%%rax"
               : "=a" (RES)
               :
               :  "%rdx");
  return result;
}
```

Benchmarking pitfalls

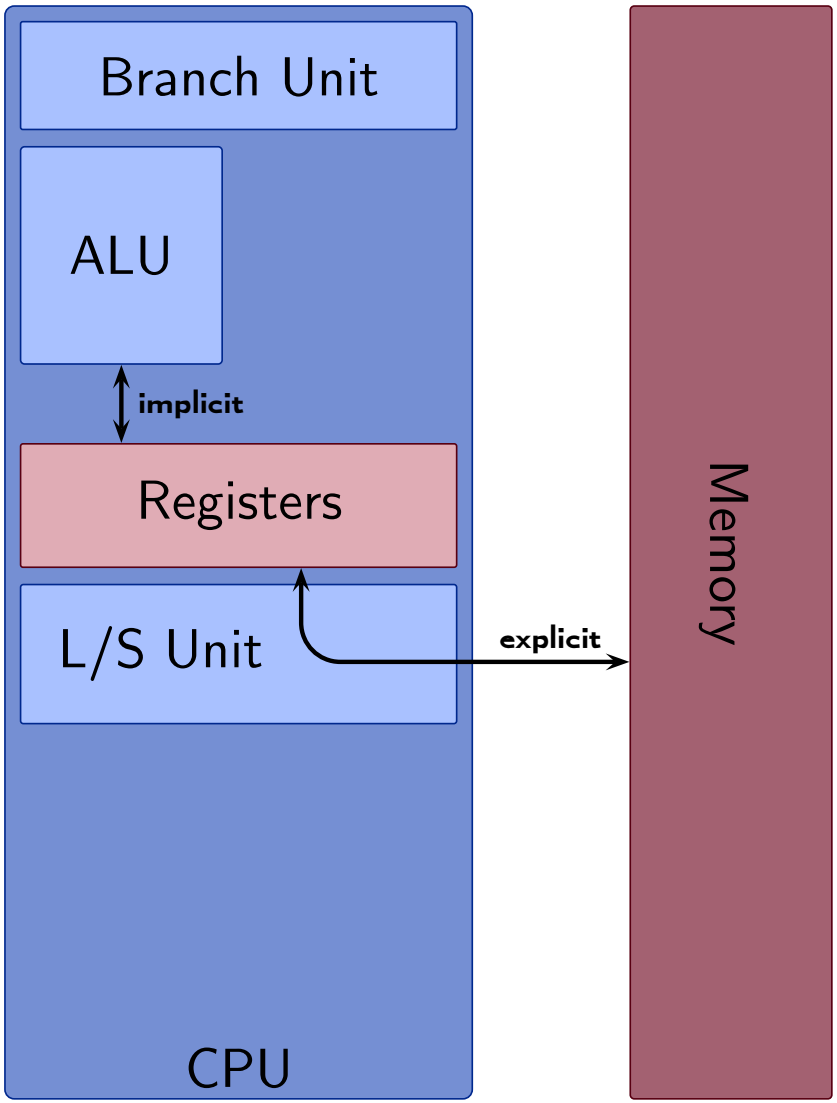
1. Your program is not running exclusively on the CPU, there may be interrupts
   Solution: Measure many times, take the *median* (not average!)
   Remark: Also report quartiles
2. The `rdtsc` instruction reports *reference* cycles, your CPU may run at a different speed
   Solution: Switch off frequency scaling and TurboBoost/TurboCore
3. Hyperthreading may run another process on the same physical core as your program
   Solution: Switch off hyperthreading
4. Getting reproducible, publicly verifiable benchmarks is hard
   Solution: Use public benchmarking framework SUPERCOP by Bernstein and Lange:

   `http://bench.cr.yp.to`

   Remark: Please submit cryptographic software to eBACS!

Computers and computer programs

A highly simplified view



- A program is a sequence of *instructions*
- Load/Store instructions move data between memory and registers (processed by the L/S unit)
- Branch instructions (conditionally) jump to a position in the program
- Arithmetic instructions perform simple operations on values in registers (processed by the ALU)
- Registers are fast (fixed-size) storage units, addressed "by name"

A first program
Adding up 1000 integers

1. Set register R1 to zero
2. Set register R2 to zero
3. Load 32-bits from address START+R2 into register R3
4. Add 32-bit integers in R1 and R3, write the result in R1
5. Increase value in register R2 by 4
6. Compare value in register R2 to 4000
7. Goto line 3 if R2 was smaller than 4000

A first program

Adding up 1000 integers in readable syntax

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
result += tmp
ctr += 4
unsigned<? ctr - 4000
goto looptop if unsigned<
```

Running the program

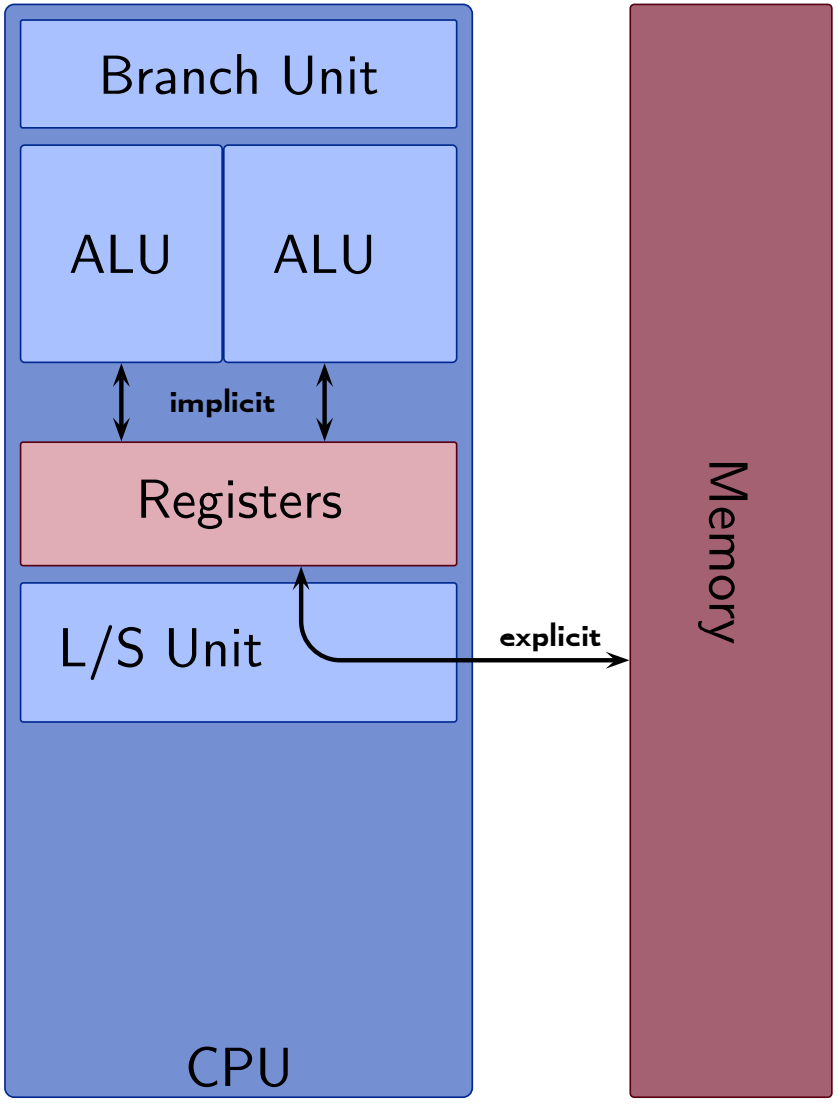
- Easy approach: Per "time-slot" (*cycle*) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction
- Other approach: Chop instructions into smaller tasks, e.g. for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
- Overlap instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- This is called pipelined execution (many more stages possible)
- Advantage: cycles can be much shorter (higher *clock speed*)
- Requirement for overlapping execution: instructions have to be independent

Instruction throughput and latency

- While the ALU is executing an instruction the L/S and branch units are idle
- Idea: Duplicate fetch and decode, handle two or three instructions per cycle
- While we're at it: Why not deploy two ALUs
- This concept is called *superscalar* execution
- Number of independent instructions of one type per cycle: **throughput**
- Number of cycles that need to pass before the result can be used: **latency**

An example computer

Still highly simplified



Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

Adding up 1000 integers on this computer

- Need at least 1000 load instructions: ≥ 1000 cycles
- Need at least 999 addition instructions: ≥ 500 cycles
- At least 1999 instructions: ≥ 500 cycles
- **Lower bound**: 1000 cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

How about our program?

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
result += tmp
ctr += 4
unsigned<? ctr - 4000
goto looptop if unsigned<
```

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
# wait 2 cycles for tmp
result += tmp
ctr += 4
# wait 1 cycle for ctr
unsigned<? ctr - 4000
```

- Addition has to wait for load
- Comparison has to wait for addition
- Each iteration of the loop takes 8 cycles
- Total: > 8000 cycles
- **This program sucks!**

Making the program fast

Step 1 – Unrolling

```
result = 0
tmp = mem32[START+0]
result += tmp
tmp = mem32[START+4]
result += tmp
tmp = mem32[START+8]
result += tmp

...

tmp = mem32[START+3996]
result += tmp
```

```
result = 0
tmp = mem32[START+0]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+4]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+8]
# wait 2 cycles for tmp
result += tmp

...

tmp = mem32[START+3996]
```

- Remove all the loop control: *unrolling*
- Each load-and-add now takes 3 cycles
- Total: ≈ 3000 cycles
- Better, but still too slow

Making the program fast

Step 2 – Instruction Scheduling

```
result = mem32[START + 0]
tmp0   = mem32[START + 4]
tmp1   = mem32[START + 8]
tmp2   = mem32[START +12]

result += tmp0
tmp0 = mem32[START+16]
result += tmp1
tmp1 = mem32[START+20]
result += tmp2
tmp2 = mem32[START+24]

...

result += tmp2
tmp2 = mem32[START+3996]
result += tmp0
result += tmp1
result += tmp2
result = mem32[START + 0]
tmp0   = mem32[START + 4]
tmp1   = mem32[START + 8]
tmp2   = mem32[START +12]
result += tmp0
tmp0 = mem32[START+16]
# wait 1 cycle for result
```

- Load values earlier
- Load latencies are hidden
- Use more registers for loaded values (`tmp0`, `tmp1`, `tmp2`)
- Get rid of one addition to zero
- Now arithmetic latencies kick in

Making the program fast

Step 3 – More Instruction Scheduling (two accumulators)

```
result0 = mem32[START + 0]
tmp0    = mem32[START + 8]
result1 = mem32[START + 4]
tmp1    = mem32[START +12]
tmp2    = mem32[START +16]

result0 += tmp0
tmp0 = mem32[START+20]
result1 += tmp1
tmp1 = mem32[START+24]
result0 += tmp2
tmp2 = mem32[START+28]

...

result0 += tmp1
tmp1 = mem32[START+3996]
result1 += tmp2
result0 += tmp0
result1 += tmp1
result0 += result1
```

- Use one more accumulator register (`result1`)
- All latencies hidden
- Total: 1004 cycles
- Asymptotically n cycles for n additions

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (**instruction scheduling**)
  - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers
- Opposing requirements to **register allocation** (assigning registers to live variables, minimizing memory access)
- Both instruction scheduling and register allocation are NP hard
- So is the joint problem
- Many instances are efficiently solvable

Architectures and microarchitectures

What instructions and how many registers do we have?

- Instructions are defined by the **instruction set**
- Supported register names are defined by the **set of architectural registers**
- Instruction set and set of architectural registers together define the **architecture**
- Examples for architectures: x86, AMD64, ARMv6, ARMv7, UltraSPARC
- Sometimes base architectures are extended, e.g., MMX, SSE, NEON

What determines latencies etc?

- Different **microarchitectures** implement an architecture
- Latencies and throughputs are specific to a microarchitecture
- Example: Intel Core 2 Quad Q9550 implements the AMD64 architecture

Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
- Typically requires more physical than architectural registers and **register renaming**
- Harder for the (assembly) programmer to understand what exactly will happen with the code
- Harder to come up with optimal scheduling
- Harder to screw up completely

Optimizing Crypto vs. optimizing "something"

- So far there was nothing crypto-specific in this lecture
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (e.g., keys)
- Information about secret data must not leak through side channels
- Most critical for software implementations on "large" CPUs: software must take constant time (independent of secret data)

Timing leakage part I

- Consider the following piece of code:

  if s **then**
  $r \leftarrow A$
  else
  $r \leftarrow B$
  end if

- General structure of any conditional branch
- A and B can be large computations, r can be a large state
- This code takes different amount of time, depending on s
- Obvious timing leak if s is secret
- Even if A and B take the same amount of cycles this is *generally not* constant time!
- Reasons: Branch prediction, instruction-caches
- **Never use secret-data-dependent branch conditions**

Eliminating branches

- So, what do we do with this piece of code?

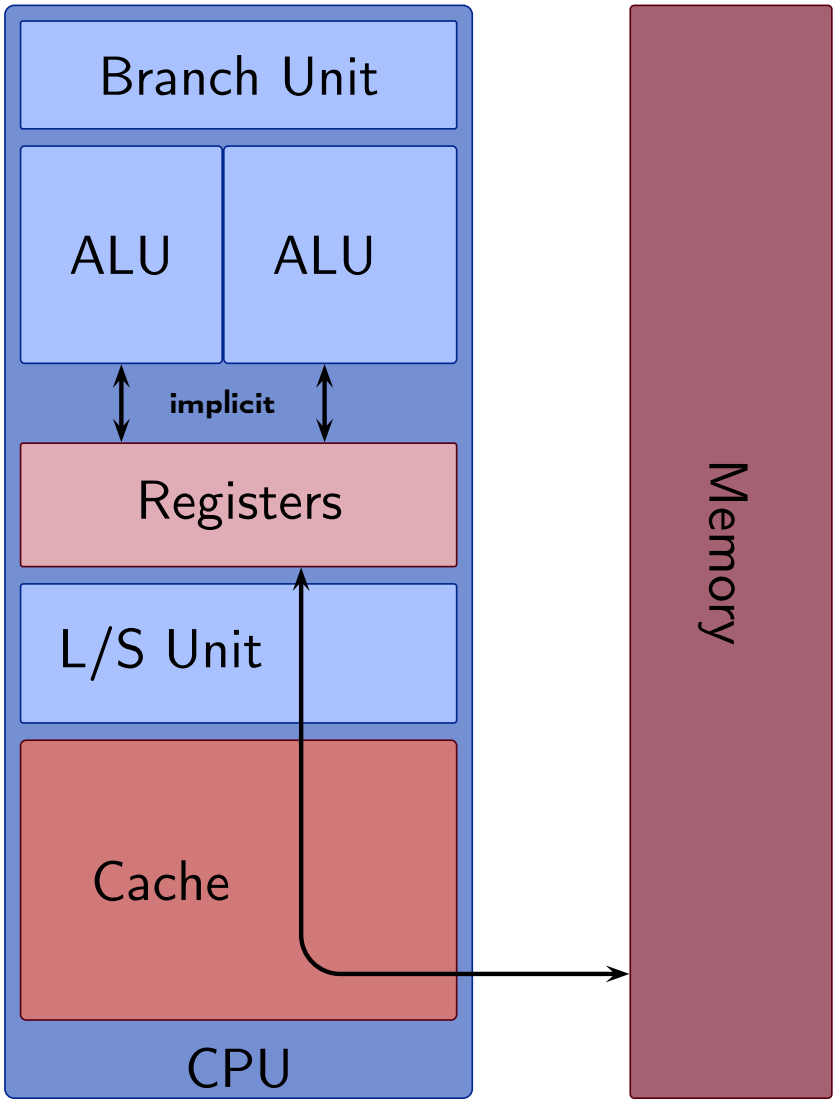
  if s **then**
  $r \leftarrow A$
  else
  $r \leftarrow B$
  end if

- Replace by

$$r \leftarrow sA + (1 - s)B$$

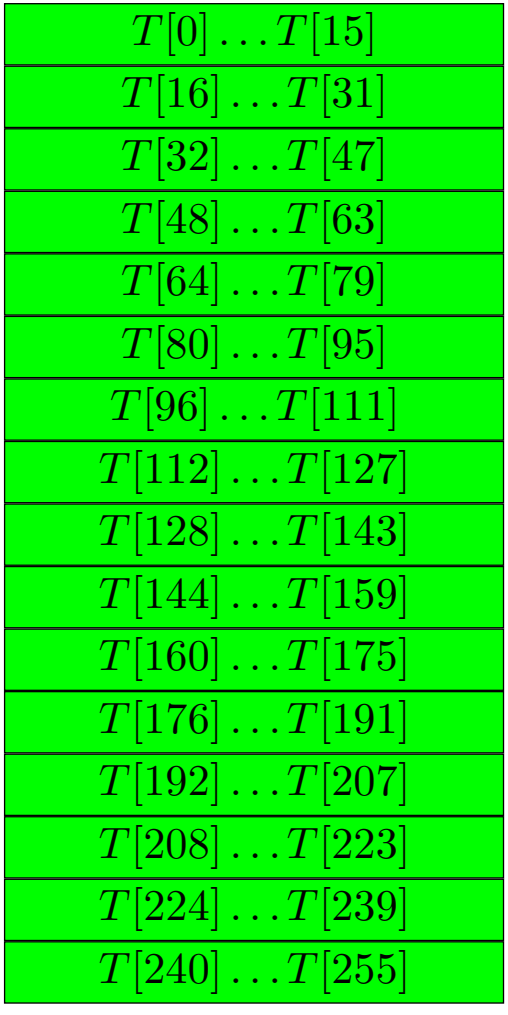
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Cached memory access



- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data is not in the cache (**cache miss**)

Timing leakage part II



Some comments on cache-timing

- This is only the *most basic* cache-timing attack
- Non-secret cache lines are not enough for security
- Load/Store addresses influence timing in many different ways
- **Do not access memory at secret-data-dependent addresses**
- Timing attacks are practical:
  Osvik, Tromer, Shamir, 2006: 65 ms to steal a 256-bit AES key used for Linux hard-disk encryption
- *Remote* timing attacks are practical:
  Brumley, Tuveri, 2011: A few minutes to steal ECDSA signing key from OpenSSL implementation

Eliminating lookups

- Want to load item at (secret) position p from table of size n
- Load all items, use arithmetic to pick the right one:

  for i from 0 to $n-1$ **do**
  $d \leftarrow T[i]$
  if $p = i$ **then**
  $r \leftarrow d$
  end if
  end for

- Problem 1: if-statements are not constant time (see before)
- Problem 2: Comparisons are not constant time, replace by, e.g.:

```
static unsigned long long eq(uint32_t a, uint32_t b)
{
  unsigned long long t = a ^ b;
  t = (-t) >> 63;
  return 1-t;
}
```

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- Test this with `valgrind` and *uninitialized secret data* (or use Langley's `ctgrind`)

"In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs.)"

—Langley, Apr. 2010

"So the argument to the `DIV` instruction was smaller and `DIV`, on Intel, takes a variable amount of time depending on its arguments!"

—Langley, Feb. 2013

Dangerous arithmetic (examples)

- `DIV`, `IDIV`, `FDIV` on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (`FSIN`, `FCOS`...)
- `MUL`, `MULHW`, `MULHWU` on many PowerPC CPUs
- `UMULL`, `SMULL`, `UMLAL`, and `SMLAL` on ARM Cortex-M3.

Solution

- Avoid these instructions
- Make sure that inputs to the instructions don't leak timing information

The software assignment

Background

Writing crypto software

1. Start with slow, potentially insecure, but functioning reference implementation in C
2. Remove main sources for timing leakage, i.e.,
   - remove secret-dependent branches
   - remove secretly indexed memory access
3. Profile the code, optimize most important routines
4. Typically use assembly for (micro-)architecture specific optimization

Typical minimal building blocks

1. Elliptic-curve Diffie-Hellman (ECDH) for key exchange
2. Some streamcipher for bulk data encryption
3. Some symmetric authenticator (MAC)

The assignment

- Given C reference implementations of
  - ChaCha20 stream cipher,
  - Poly1305 authenticator, and
  - ECDH on Curve25519 in Edwards form,
- produce optimized implementations for the ARM Cortex-M4

For details see `sw-assignment-2019.pdf` in Brightspace or at
https://cryptojedi.org/peter/teaching/ce2019/sw-assignment-2019.pdf

Preparation for next week

https://github.com/joostrijneveld/STM32-getting-started