TU WIEN Informatics

Systems Computing Embedded

Advanced Computer Architecture

Lab2: High-Level Synthesis Lab

Johannes Kappes, Daniel Mueller-Gritschneder

18.11.2025

High-Level Synthesis Lab Structure

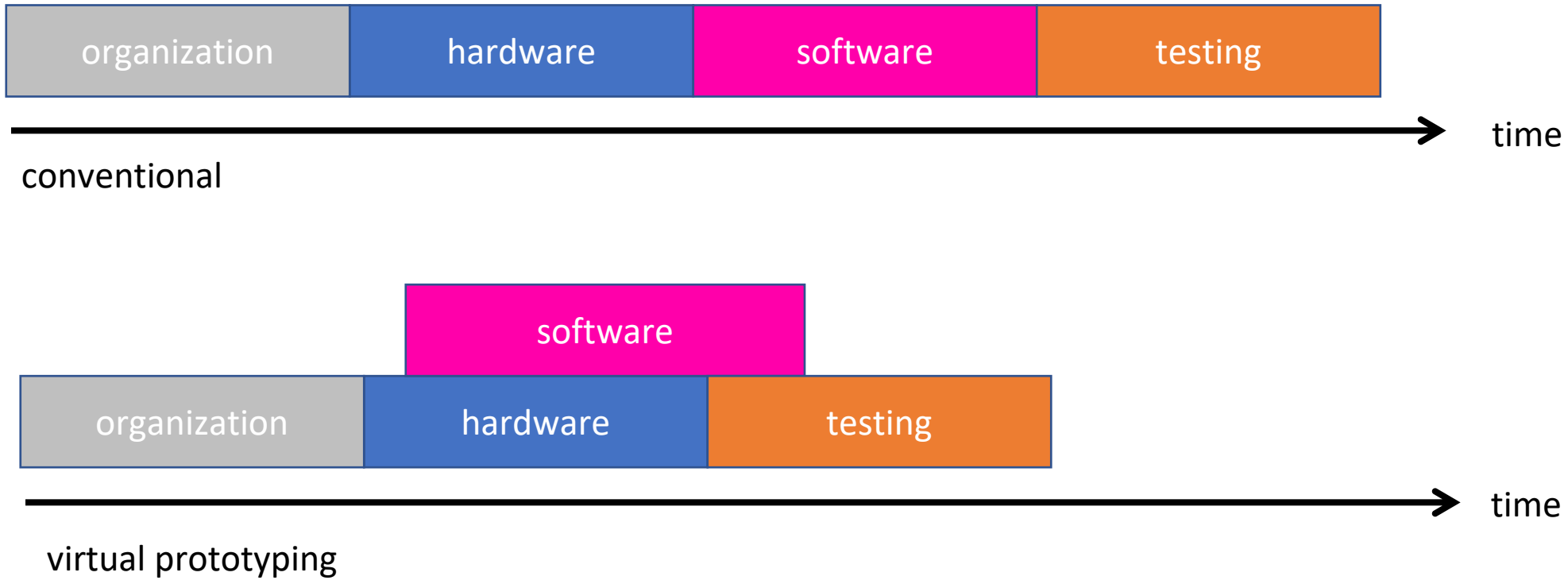
3 Modules:

- M1 „Hello World!“ on a VP
- M2 Introduction into High Level Syntesis Tool (Vivado HLS)
- M3 Acceleration of Advanced Encryption Standard (AES) algorithm

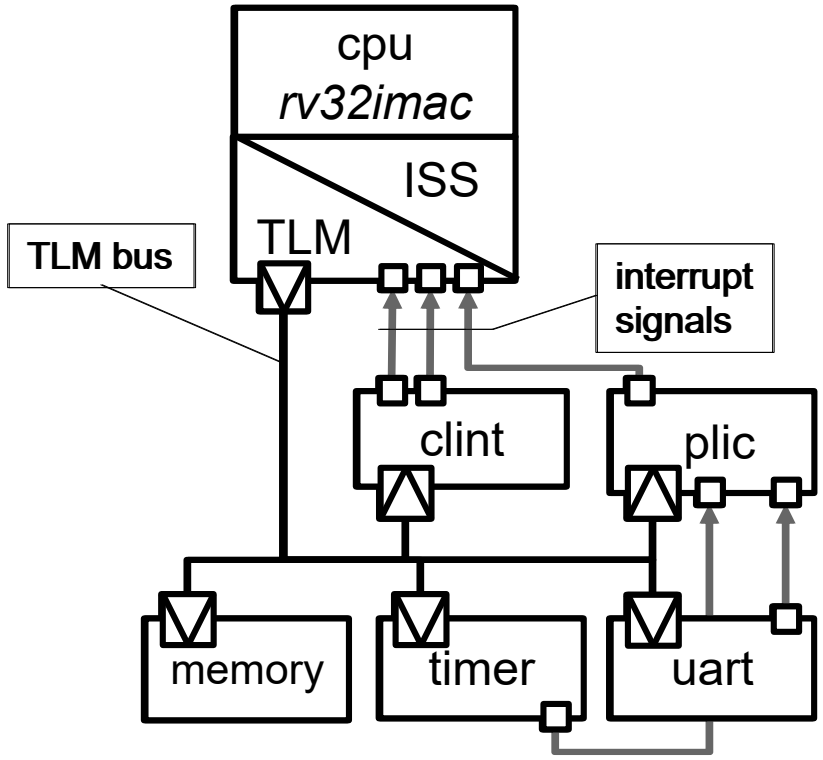
Module 1: Hello World! On a Virtual Prototype

Module 1.1: Background

Why Virtual Prototyping?



EdaDuino - SoC (M1)



- 32-bit RISC-V CPU
- Core-local Interrupt Controller (clint) with Real-time Clock
  - (Machine Timer Interrupt)
  - MSI (Multi-Core)

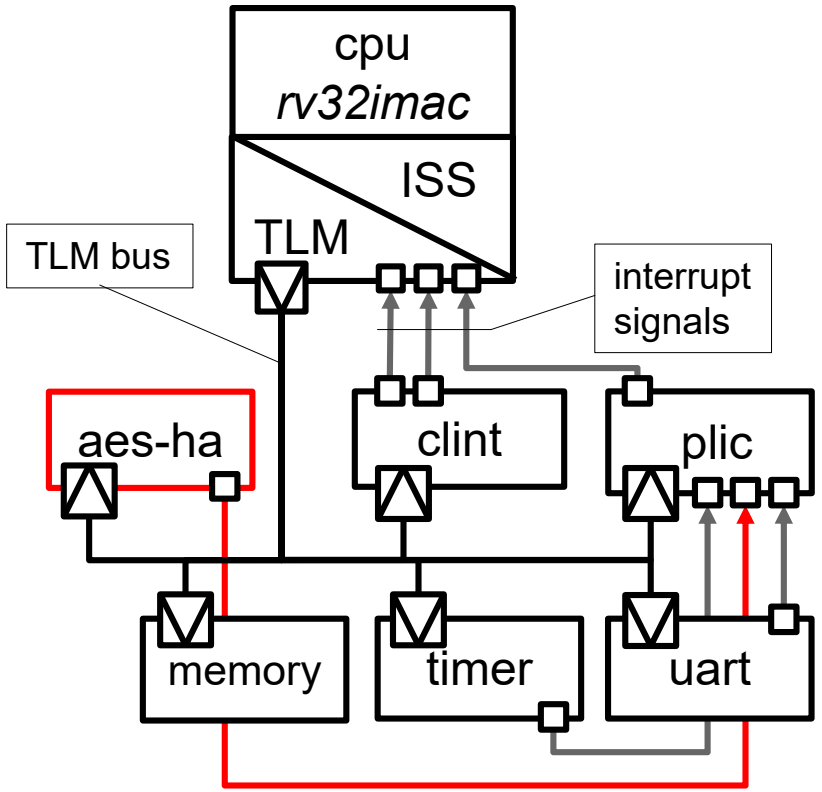
- Platform-level Interrupt Controller (plic) as Interrupt MUX for:

  - UART with Interrupt

  - Timer with Interrupt

- Transaction Level Modeling (TLM) of Memory bus

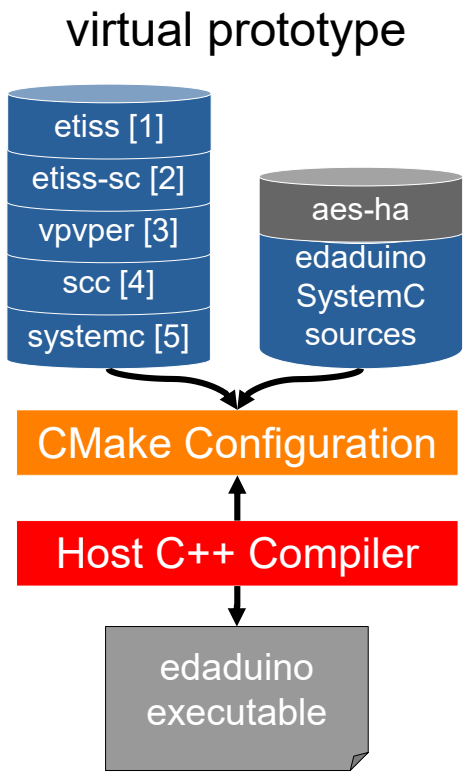
EdaDuino - SoC (M3)



- AES Hardware Accelerator (generated with Vivado-HLS)

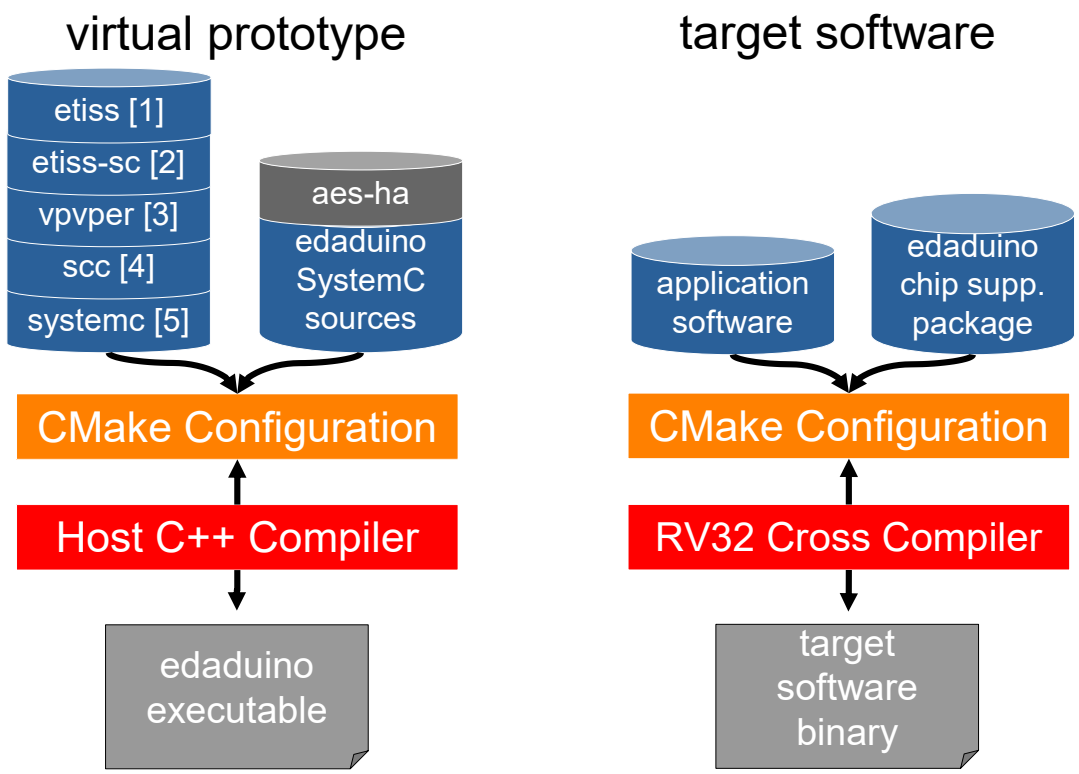
- 32-bit RISC-V CPU
- Core-local Interrupt Controller (clint) with Real-time Clock
  - (Machine Timer Interrupt)
  - MSI (Multi-Core)
- Platform-level Interrupt Controller (plic) as Interrupt MUX for:
  - UART with Interrupt
  - Timer with Interrupt
- Transaction Level Modeling (TLM) of Memory bus

EdaDuino - Toolchain / Build System

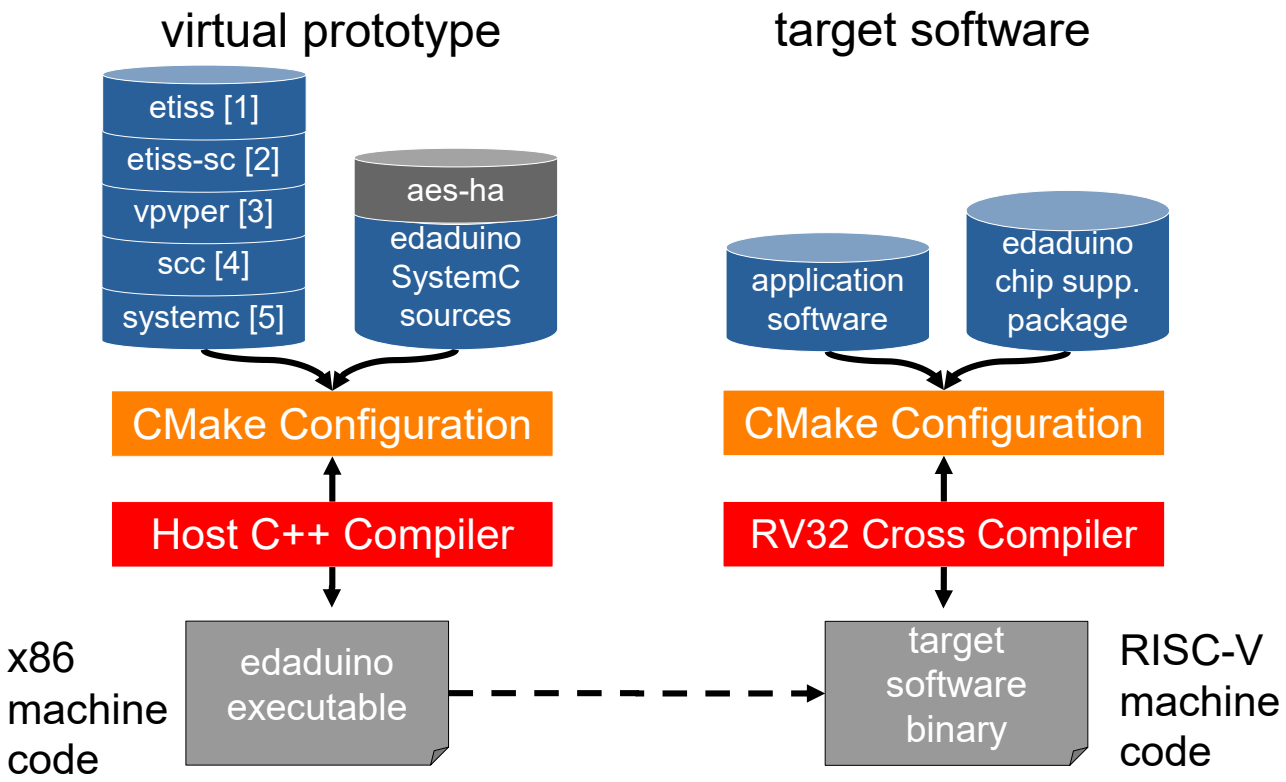


[1] https://github.com/tum-ei-eda/etiss
[2] https://github.com/tum-ei-eda/etiss-sc
[3] https://github.com/VP-Vibes/VPV-Peripherals
[4] https://github.com/VP-Vibes/SystemC-Components
[5] https://github.com/accellera-official/systemc

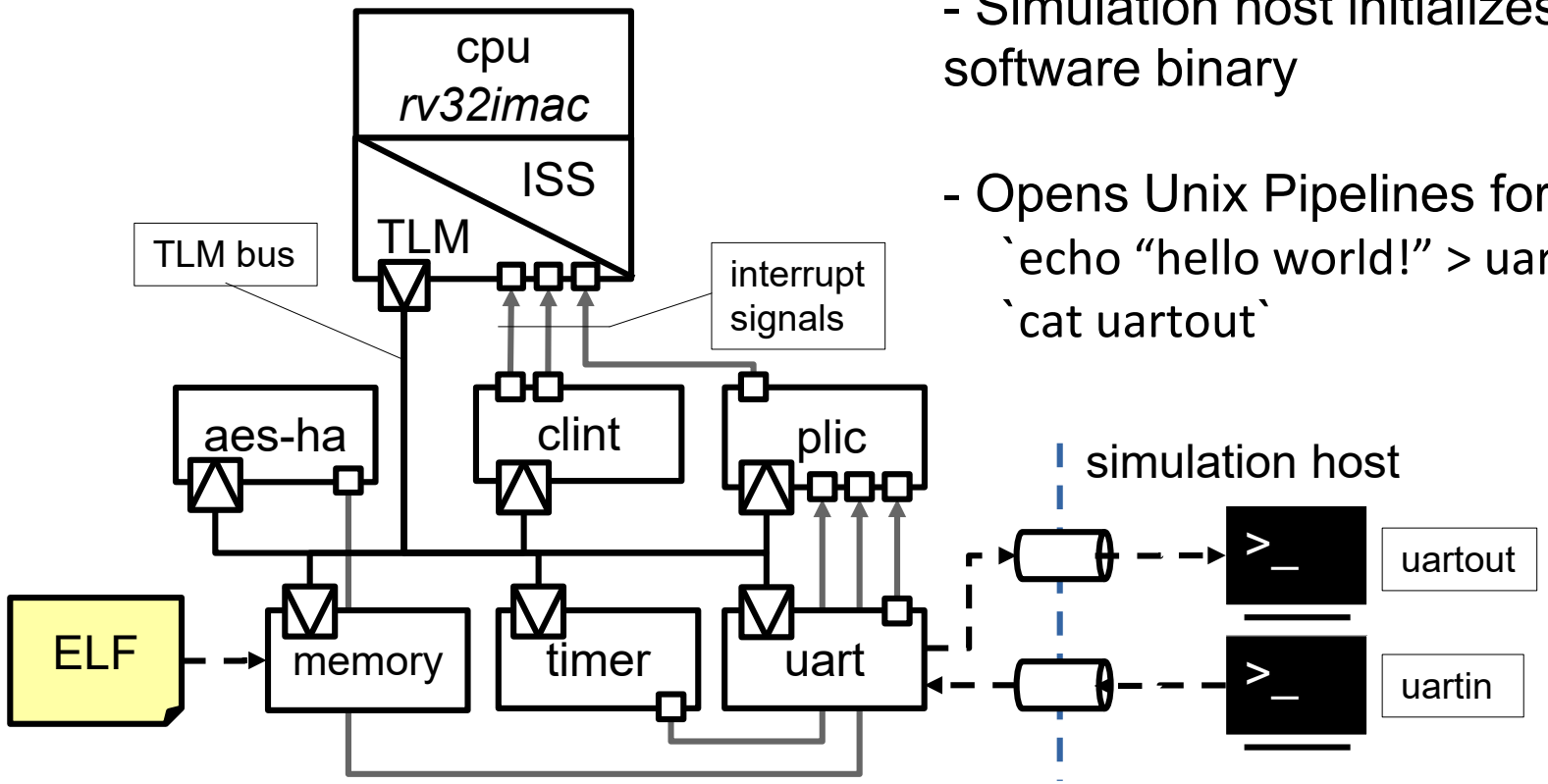
EdaDuino - Toolchain / Build System



EdaDuino - Toolchain / Build System



EdaDuino - Virtual Platform



- Simulation host initializes memory with target software binary

- Opens Unix Pipelines for UART terminal, e.g.,
  `echo “hello world!” > uartin`
  `cat uartout`

M1.2 Task: Utilizing UART Interrupt

Polling vs. Interrupt-driven Hardware synchronization

- **Polling**:
  - cyclical sampling a peripherals status by an application program (synchronous)
  - cyclical checks can be scheduled
    - continuous sampling: “busy-wait”:
    - increase sampling period: better performance, risk of losing events

- **Interrupt-driven**:
  - peripheral reports a (configured) state change event to the CPU by and interrupt

The System

- UART Peripheral *
  - 16750 standard
  - memory mapped registers
  - single Interrupt line to CPU

- Software:
  “Serial Mirror”: UART received data is mirrored on Transmitter data

UART Register	Address std	Addr. abs EdaDuino	Type	Access Config
RBR **THR** **DLL**	base + (0*w)	0x1000_0000	**RO** **WO** **RW**	**LCR.DLAB = lo** **LCR.DLAB = lo** **LCR.DLAB = hi**
IER **DLM**	base + (1*w)	0x1000_0004	**RW** **RW**	**LCR.DLAB = lo** **LCR.DLAB = hi**
IIR **FCR**	base + (2*w)	0x1000_0008	**RO** **WO**	
LCR	base + (3*w)	0x1000_000c	**RW**	
MCR	base + (4*w)	0x1000_0010	**RW**	
LSR	base + (5*w)	0x1000_0014	**RO**	
MSR	base + (6*w)	0x1000_0018	**RO**	
SCR	base + (7*w)	0x1000_001C	**RW**	

*elaborate guide to UART: https://www.lammertbies.nl/comm/info/serial-uart

C Driver Library

sw/csp/inc/uart_drv.h

```
/// \brief Uart control object. Holds states and registers
typedef struct Uart
{
    /* Registers */
    volatile uint8_t *rbr_; ///< Receiver Buffer Register (make sur
    volatile uint8_t *thr_; ///< Transmitter Holding Register (make
    volatile uint8_t *dll_; ///< Divisor Latch (LSByte) Register (m
    volatile uint8_t *ier_; ///< Interrupt Enable Register (make su
    volatile uint8_t *dlm_; ///< Divisor Latch (MSByte) Register (m
    volatile uint8_t *iir_; ///< Interrupt Identification Register
    volatile uint8_t *fcr_; ///< FIFO Control Register [WO]
    volatile uint8_t *lcr_; ///< Line Control Register [RW]
    volatile uint8_t *mcr_; ///< Modem Control Register [RW]
    volatile uint8_t *lsr_; ///< Line Status Register [RO]
    volatile uint8_t *msr_; ///< Modem Status Register [RO]
    volatile uint8_t *scr_; ///< Scratch [RW]
    /* Callbacks */
    void (*irq_callback)(void); ///< Function pointer to custom int
} uart_t;

void uart__set_dlab(uart_t *handle);
void uart__reset_dlab(uart_t *handle);
void uart__set_IERbit(uart_t *handle, uint8_t nbit);
void uart__reset_IERbit(uart_t *handle, uint8_t nbit);
void uart__set_irq_callback(uart_t *handle, void (*fptr)(void));
void uart__enable_interrupt(uart_t *handle, ier_bits_t interrupt);
void uart__disable_interrupt(uart_t *handle, ier_bits_t interrupt);

void uart__send_char(uart_t *handle, const char c);
char uart__get_char(uart_t *handle);
```

UART Register	Address std	Addr. abs. EdaDuino	Type	Access Config
RBR **THR** **DLL**	base + (0*w)	0x1000_0000	**RO** **WO** **RW**	**LCR.DLAB = lo** **LCR.DLAB = lo** **LCR.DLAB = hi**
IER **DLM**	base + (1*w)	0x1000_0004	**RW** **RW**	**LCR.DLAB = lo** **LCR.DLAB = hi**
IIR **FCR**	base + (2*w)	0x1000_0008	**RO** **WO**	
LCR	base + (3*w)	0x1000_000c	**RW**	
MCR	base + (4*w)	0x1000_0010	**RW**	
LSR	base + (5*w)	0x1000_0014	**RO**	
MSR	base + (6*w)	0x1000_0018	**RO**	
SCR	base + (7*w)	0x1000_001C	**RW**	

The Task

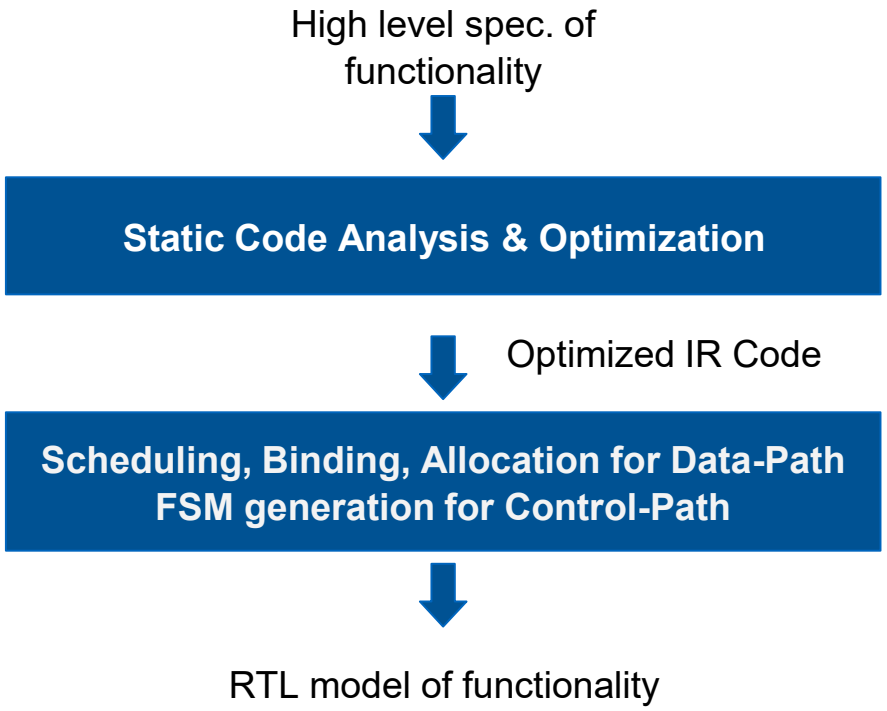
1. Write a `uart_send_string(const char* str)` driver function
2. Check the pre-implemented busy-wait “Serial Mirror”
3. Replace the busy-wait poll with a wait for interrupt (WFI)
4. Write an UART_IRQ_HANDLER
5. Check your interrupt-driven solution, if it still performs the “Serial Mirror”

Learning Goals?

- Basics of (embedded) Software Compilation Flow
- Programming Low-Level Memory Mapped I/O
- Basics of Bare-metal programming
- Basics of ETISS VP for Module part 3

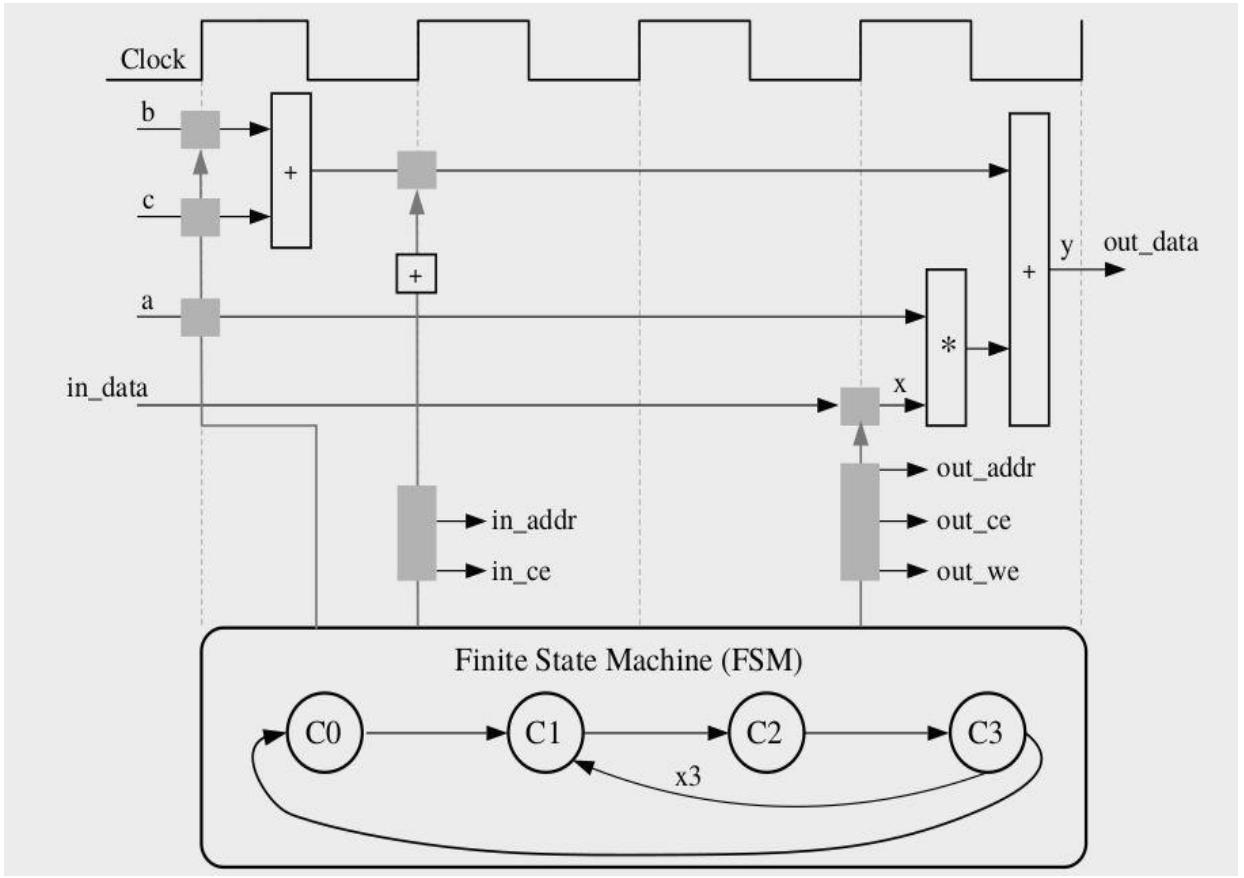
Module 2: High Level Synthesis with Vivado-HLS

High Level Synthesis (HLS)



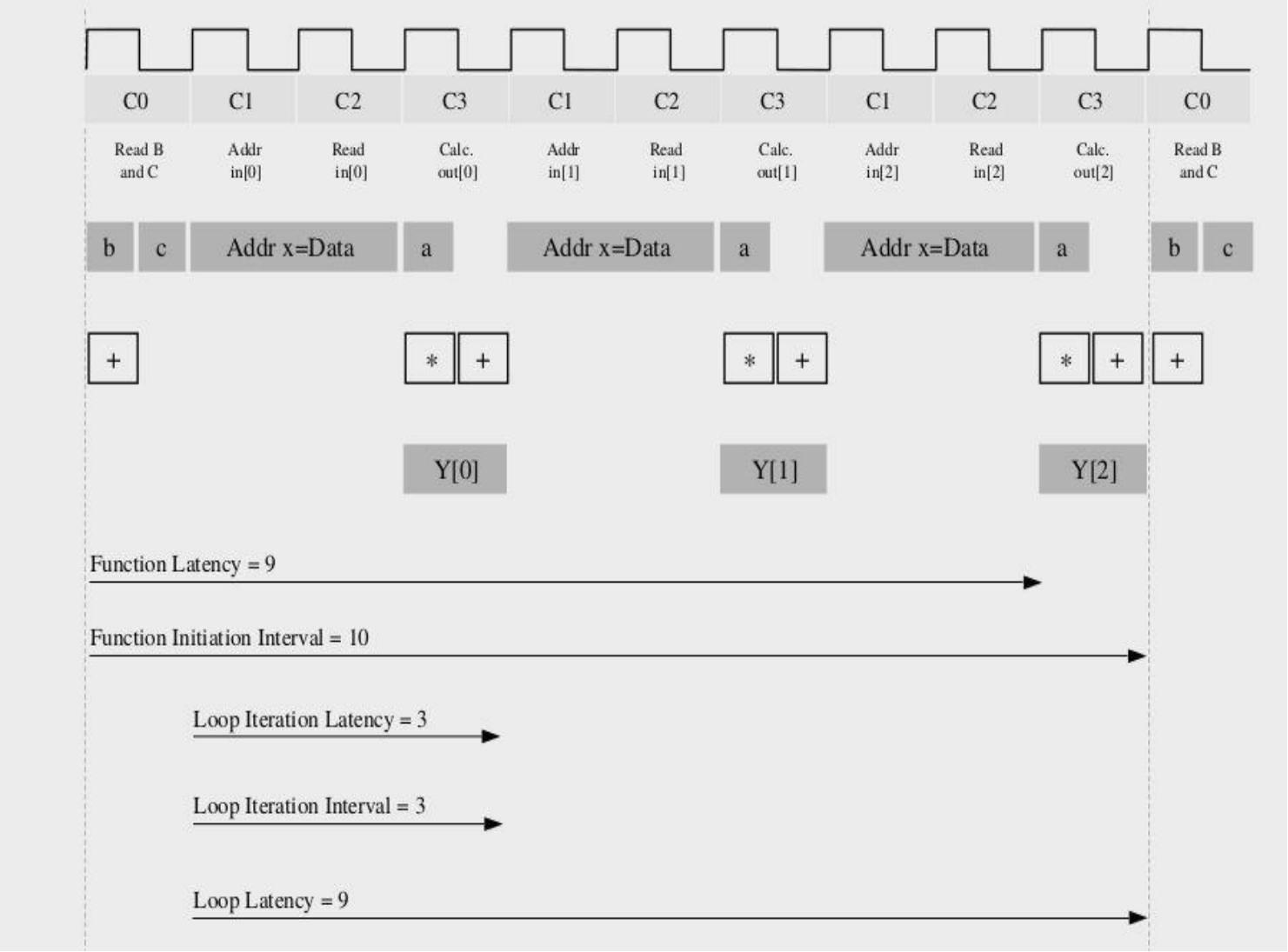
Example

```
void foo(int in[3], char a, char b, char c, int out[3]) {
  int x,y;
  for (int i=0; i<3; ++i) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```



Source: Xilinx Web

Learning Goals?



Source: Xilinx Web

HLS Benefits

- Design at higher abstraction
  – improved productivity for HW designers
    - employing Model-Based-Design principles
    - rapid system prototyping; shorter time-to-market

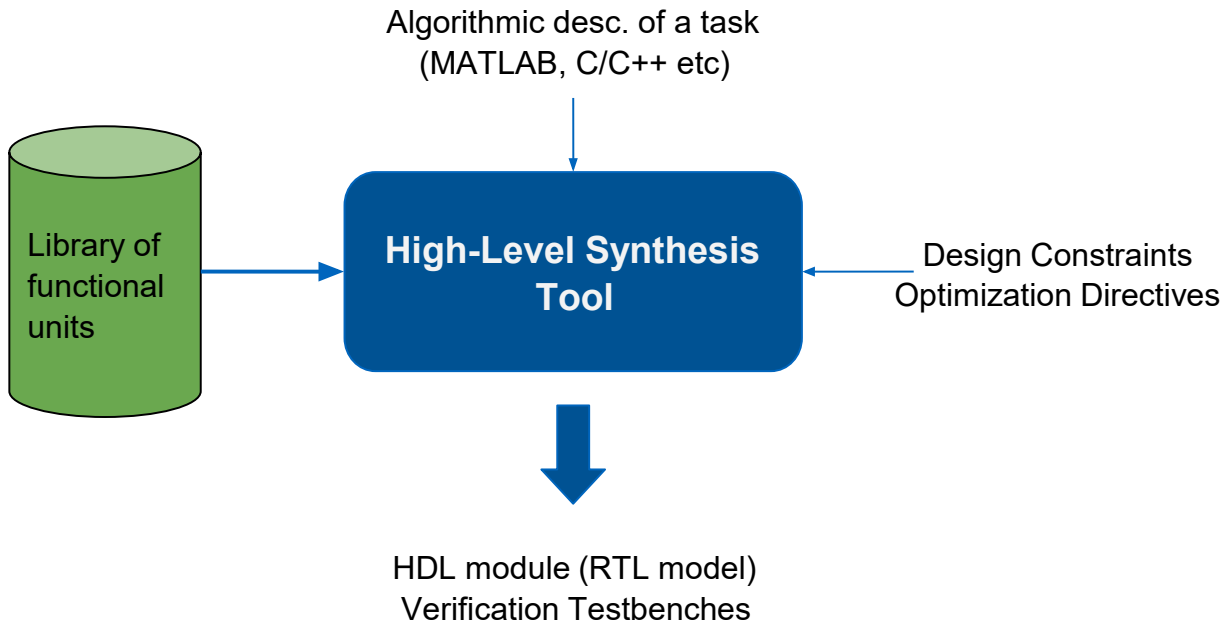
  – improved system performance for SW designers
    - HW/SW CO-design to accelerate applications

- Better Design Space Exploration
  – optimize resource, perf. trade-off early in design cycle

HLS Limitations

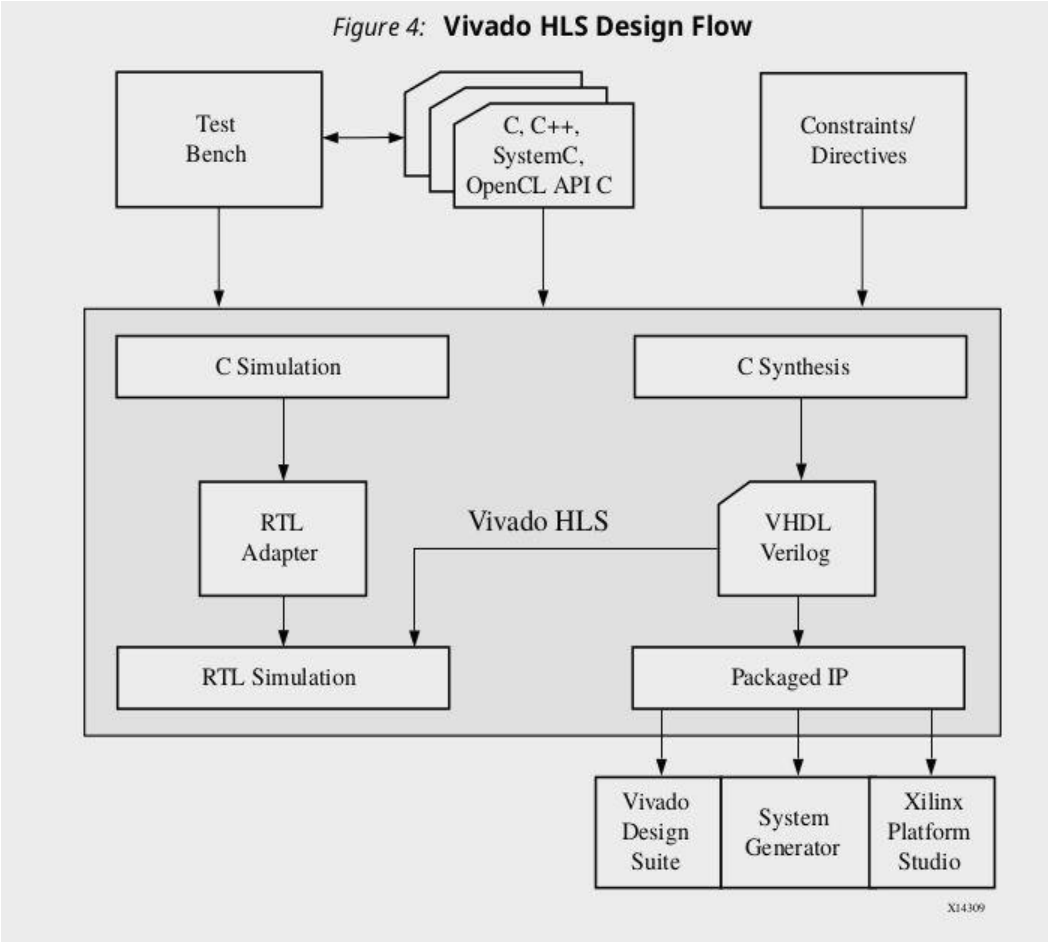
- Some SW concepts do not map well into HW
  - Datatypes: No fixed datatypes
  - Pointers: No heap for dynamic memory
  - Recursion: No notion of execution stack
  - Communication: No notion of shared memory
- Generated RTL hard to extend / modify
  - difficult to debug if RTL verification fails

HLS abstract tool view



Vivado HLS Flow

- Inputs:
  - C function
  - Constraints
  - Directives
  - C testbench
- Outputs:
  - RTL code in HDLs, SystemC etc.
  - RTL testbench
  - Report files



Source: Xilinx Web

Vivado-HLS: Implementation Aspects (1)

- Top-level function args into
  - RTL I/O ports
  - supported Interfaces (AXI)

- C functions into blocks in RTL hierarchy
  – hierarchy of sub-funcs -> hierarchy of RTL modules
  – all instances of a function use same RTL block

- Loops into blocks in RTL hierarchy with control path
  – nested-loops -> hierarchy of RTL modules

Vivado-HLS: Implementation Aspects (2)

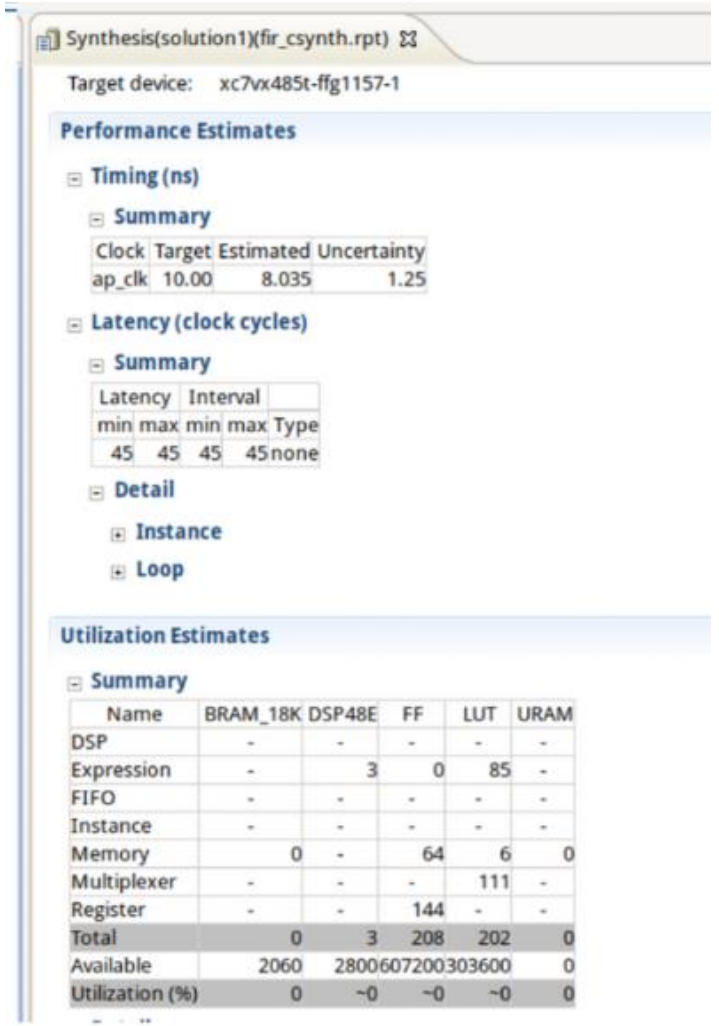
- Loops are kept rolled by default
  - RTL logic (as component) for one iteration of loop
  - all iterations scheduled in sequence using the same logic

- Arrays synthesize to Block-RAM by default
  - Arrays in I/O ports as external block-RAMs
  - FIFOs, HW Registers, Distributed RAM also possible

Synthesis

Synthesis reports:

- Estimated Timing
- Utilization
- Loop/Module:
  - iteration latency
  - initiation interval



Synthesis(solution1)(fir_csynth.rpt)

Target device: xc7vx485t-ffg1157-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.035	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
45	45	45	45	none

Detail

Instance

Loop

Utilization Estimates

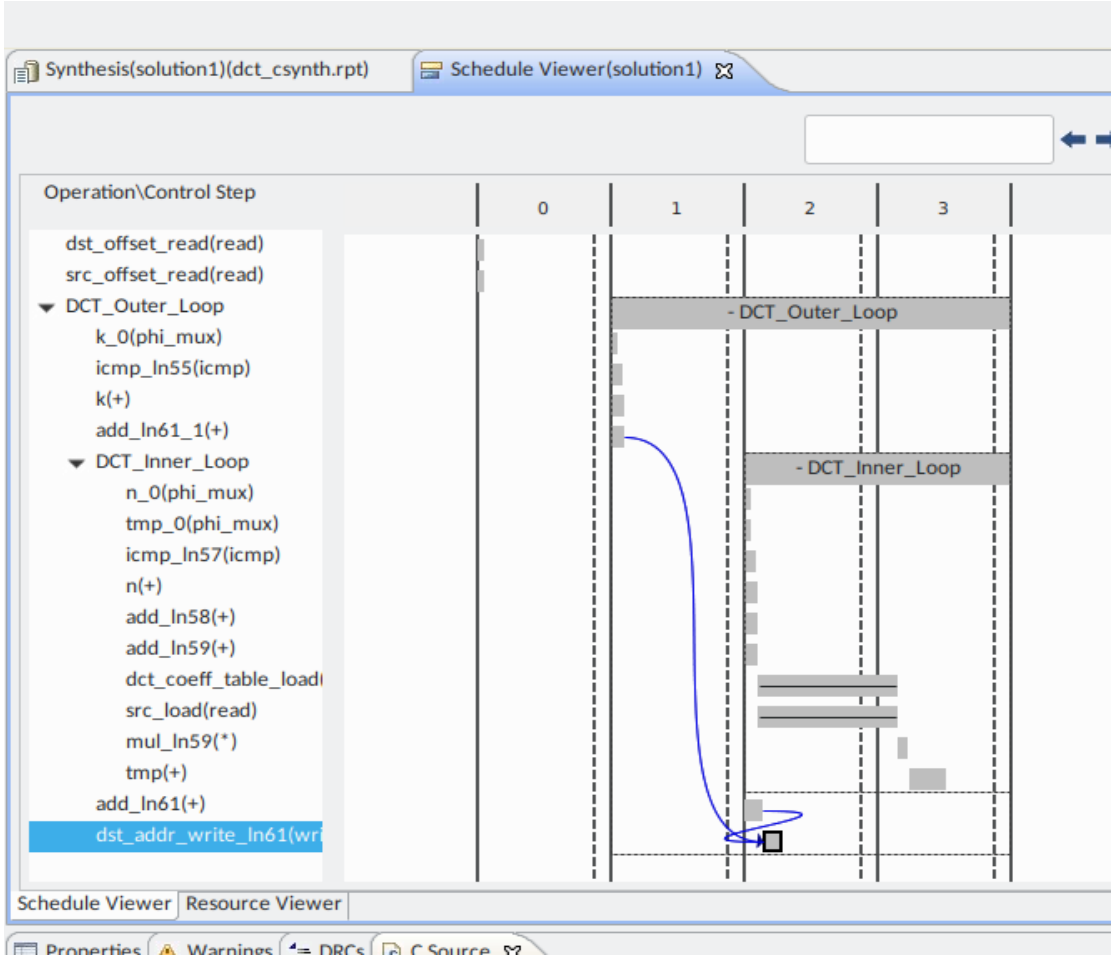
Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	3	0	85	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	0	-	64	6	0
Multiplexer	-	-	-	111	-
Register	-	-	144	-	-
Total	0	3	208	202	0
Available	2060	2800	607200	303600	0
Utilization (%)	0	~0	~0	~0	0

Analysis

Reports:

- Data Dependencies within Modules
- Various Latencies such as:
  - Loop iteration latency
  - Loop initiation interval



Optimization

- Strategies:
  - Instruct a task to execute pipelined
  - Specify latency for completion of funcs/loops
  - Specify limit on number of resources used
  - Select a specific I/O protocol for optimal integration
  - Efficient structuring of data items

Directives

Type	
Kernel Optimization	• pragma HLS allocation • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate
Interface Synthesis	• pragma HLS interface • pragma HLS protocol
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS data_pack

https://download.amd.com/docnav/documents/aem/xilinx2019_1-ug1253-sdx-pragma-reference.pdf?utm_source=chatgpt.com

M2.1 Lab tasks

Finite Impulse Response (Getting started)

- Intro to Vivado HLS tool
- Synthesize an initial solution for FIR filter (C func)
- Steps:
  - Set up an HLS project
  - Source code for FIR filter in C
  - Validate the C source
  - Create and synthesize a solution
  - Analyze the synthesized HW
  - Package the RTL as IP

HLS of a DCT Filter

- Discrete Cosine Transform (DCT) is heavily used in signal-processing applications
- Steps:
  - Start off with an initial C-level implementation
  - Analyze the performance bottlenecks
  - Optimize via directives to improve performance
  - Package the final generated IP

Module 3: Acceleration of embedded AES Cryptography with HLS

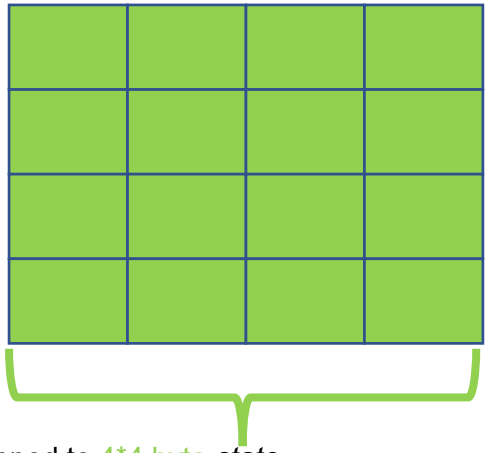
M3.1 Background

Advanced Encryption Standard (AES)

- Essentially a symmetric data-processing algorithm
  - – Block Cipher AES-128 (128-bit blocks)
  - → requires add. Block Mode of Operation (BCMO) to en-/decrypt messages of multiples blocks

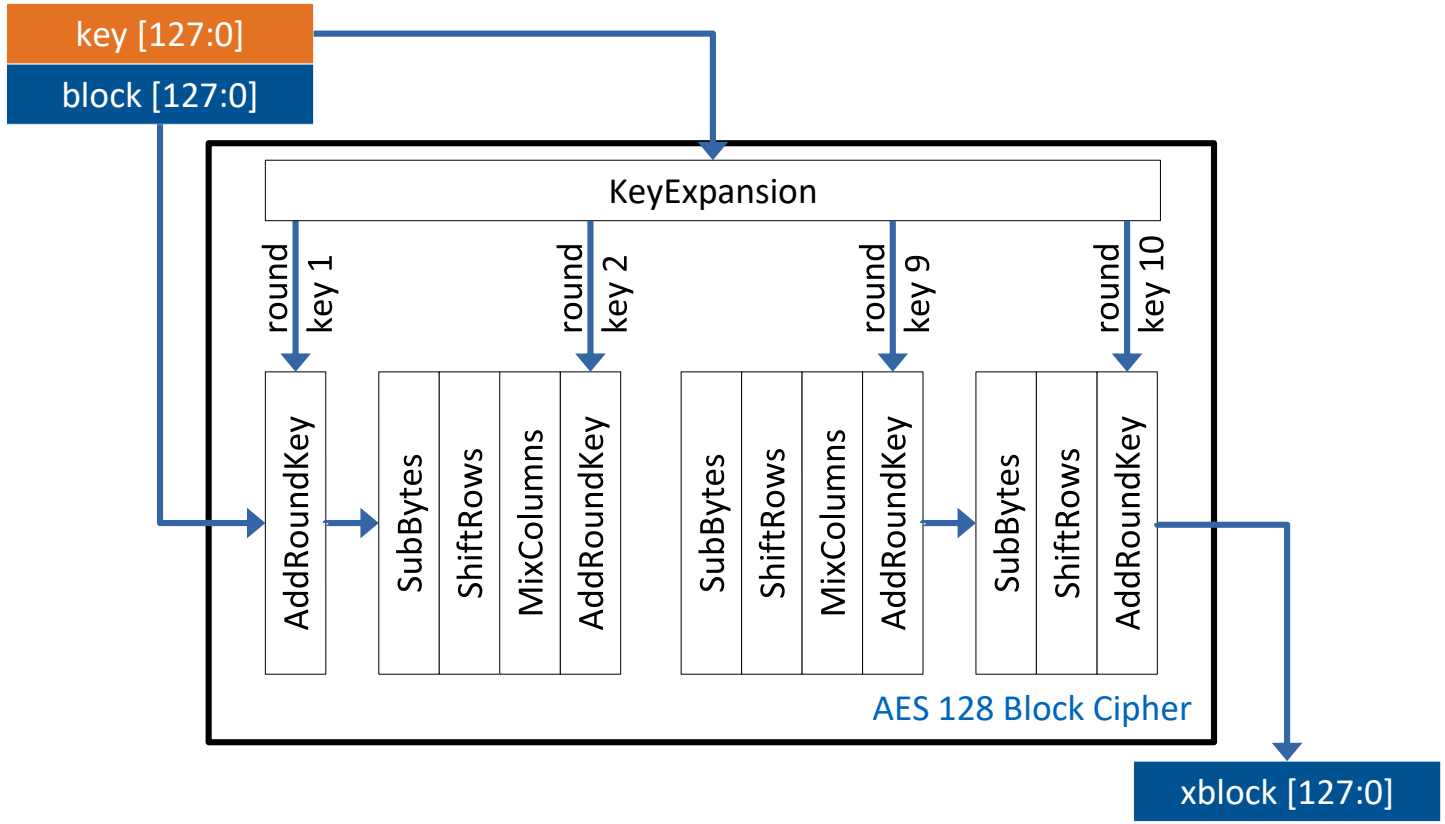
- In general compute-intensive
  - – High optimization potential for HLS
    - Regular data-flow oriented nature
    - Symmetry allows usage of same hardware for both directions
    - Please don’t do this in HLS in the real life! -> Side-Channels

AES-128

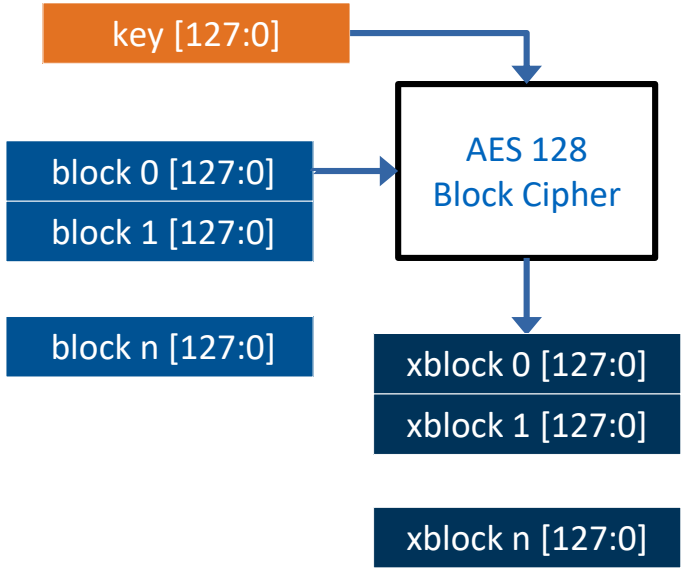


- 128-bit IO and key. Input mapped to 4*4 byte *state*
- **KeyExpansion**: Generate individual key for each round
- **AddRoundKey**: *state* is XORed with round key
- ***SubBytes***: Each *state* byte's value is substituted by a mapped value (can be a LUT or calculated) (non-linear)
- **ShiftRows**: Shifting rows of *state* matrix
- **MixColumns**: Mixing operation on columns of *state* matrix

-> computationally heavy



Encrypting a message: Straight-forward

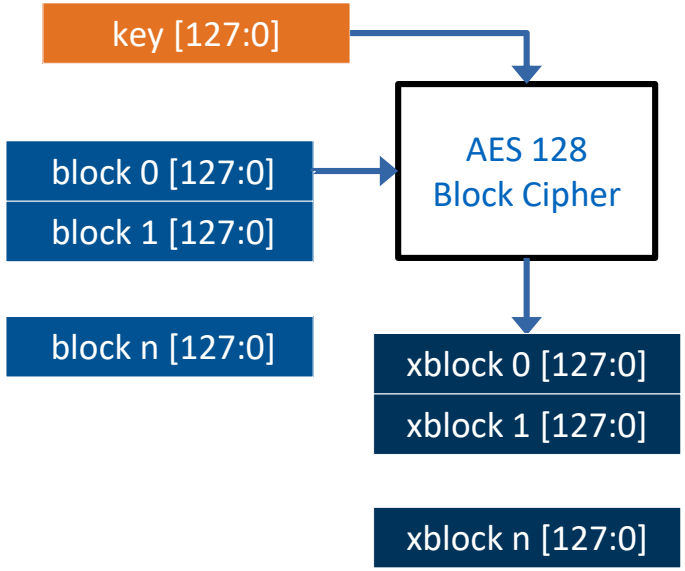


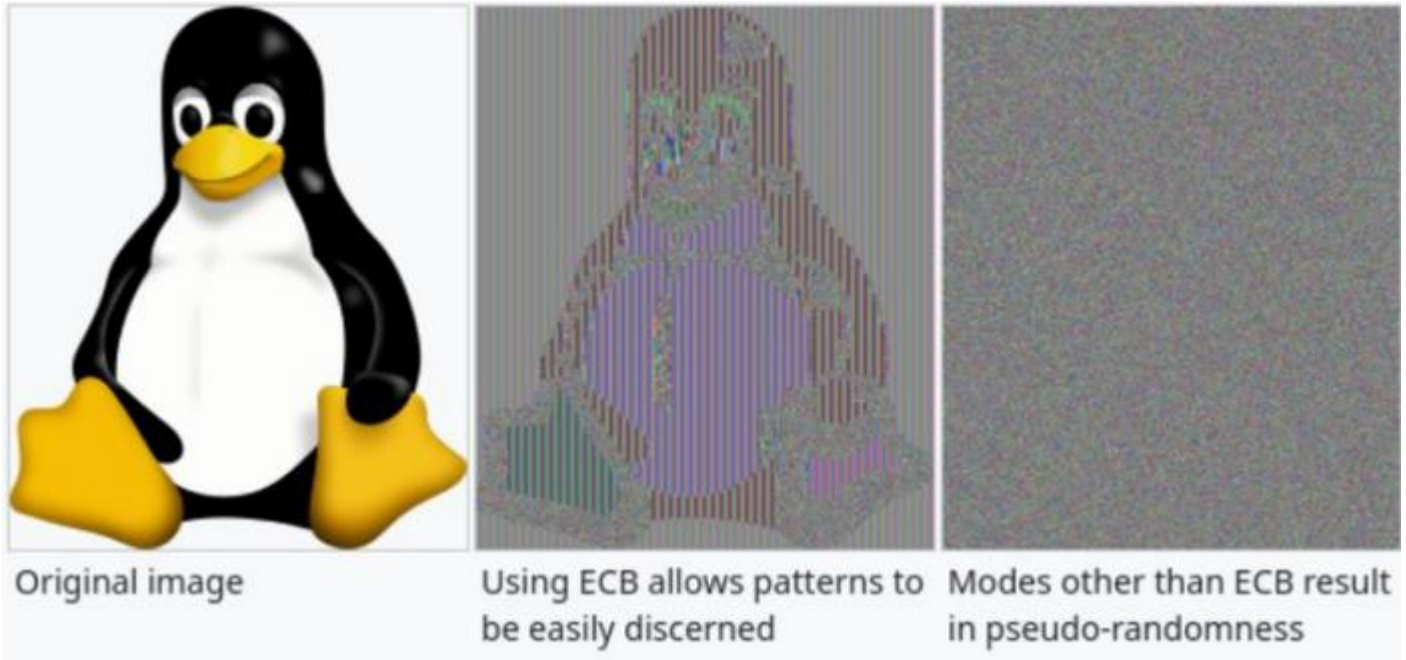
- Consider message ***m*** with *n* 128-bit blocks of information of which some bear the same information, i.e. their 128-bit value is equal

- Block-by-block transformation with AES128 would yield encrypted but recurring encrypted blocks.

—> Very bad!

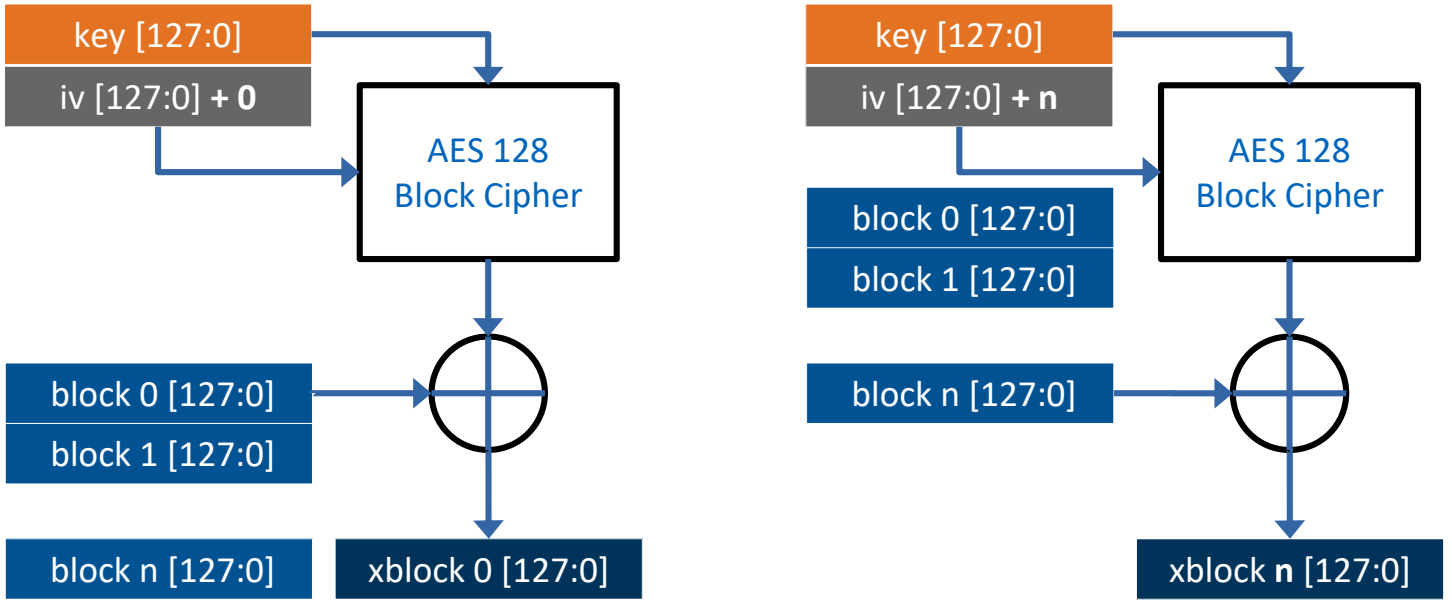
Encrypting a message: Straight-forward





Comparison between original, ECB and otherwise encrypted messages
https://upload.wikimedia.org/wikipedia/commons/thumb/9/96/Tux_encrypted_ecb.png/196px-Tux_encrypted_ecb.png

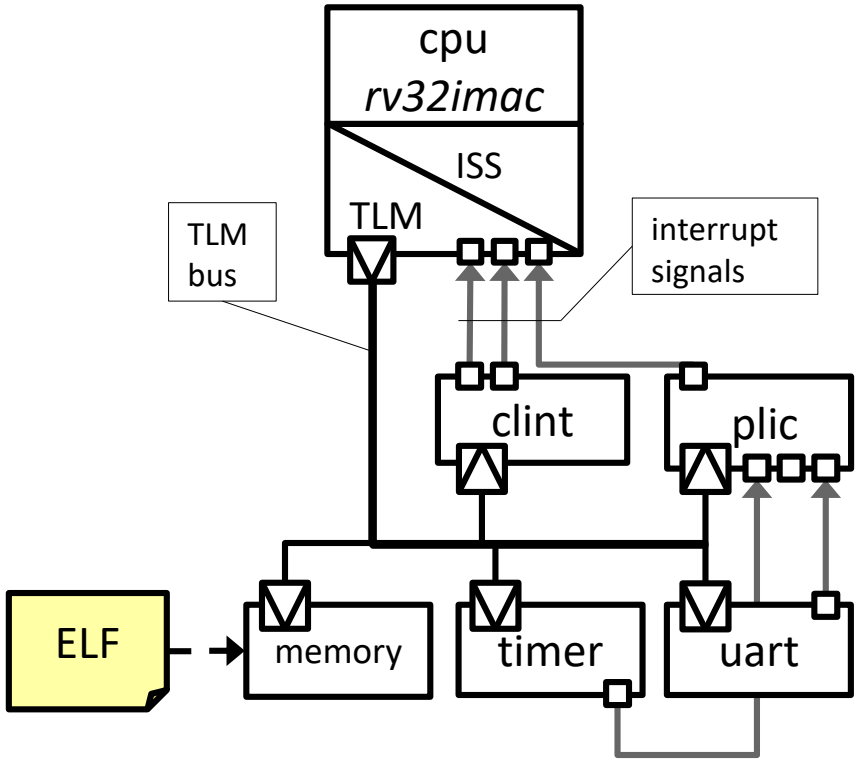
Encrypting a message: BCMO



Block Cipher Mode of Operation (BCMO) enables Block Ciphers (like AES128) to be used on multi-block messages. Here: Counter (CTR) + Initiation Vector (IV)

M3.2 Task 1: AES-SW

Task 1: Pure Software implementation



- Decrypting a message ***m,*** containing a secret ASCII-based picture.
- ***m*** can be resembled by a 46x192 byte = 46x(12x128) bit matrix.
- ***m***, private key (AES key) and public key (IV) are provided

Task: Application Profiling the AES software library as a software implementation on EDAduino. **#cycles spent for this solution?**

M3.2 Task 2: AES-HA

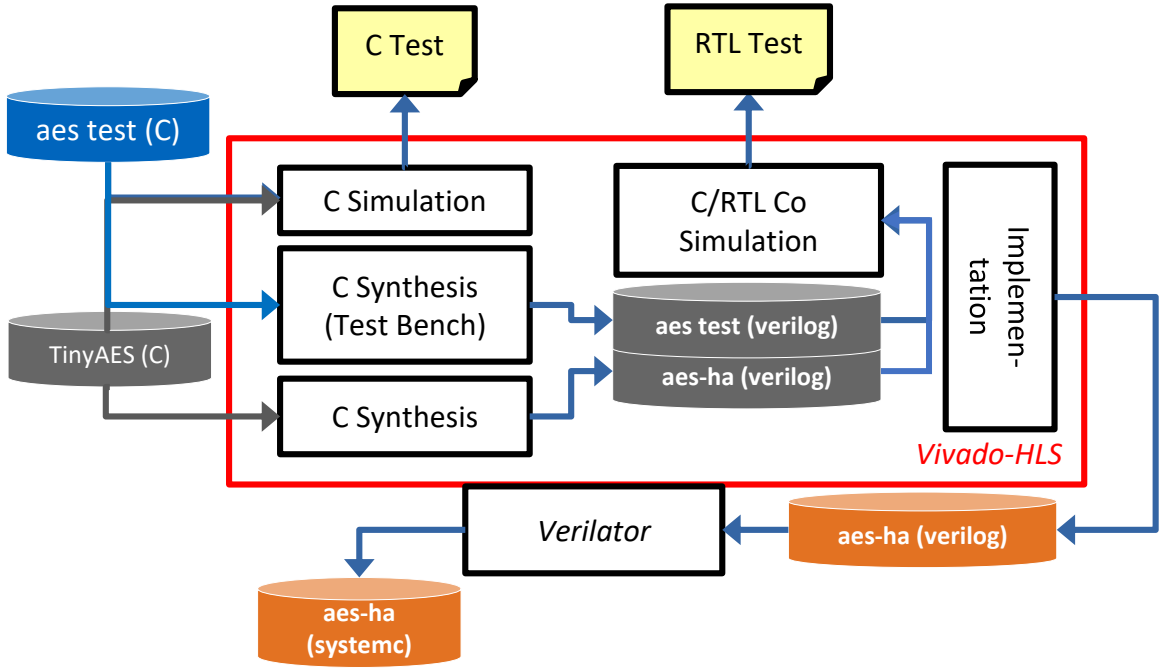
Extended Design Flow: AES-CTR to AES-HA

1) AES-CTR (AES128 w/ CTR BCMO) as C software library

2) Vivado-HLS builds RTL (normally we would export to a Xilinx FPGA design suite)

3) Verilator builds cycle-accurate SystemC model from RTL

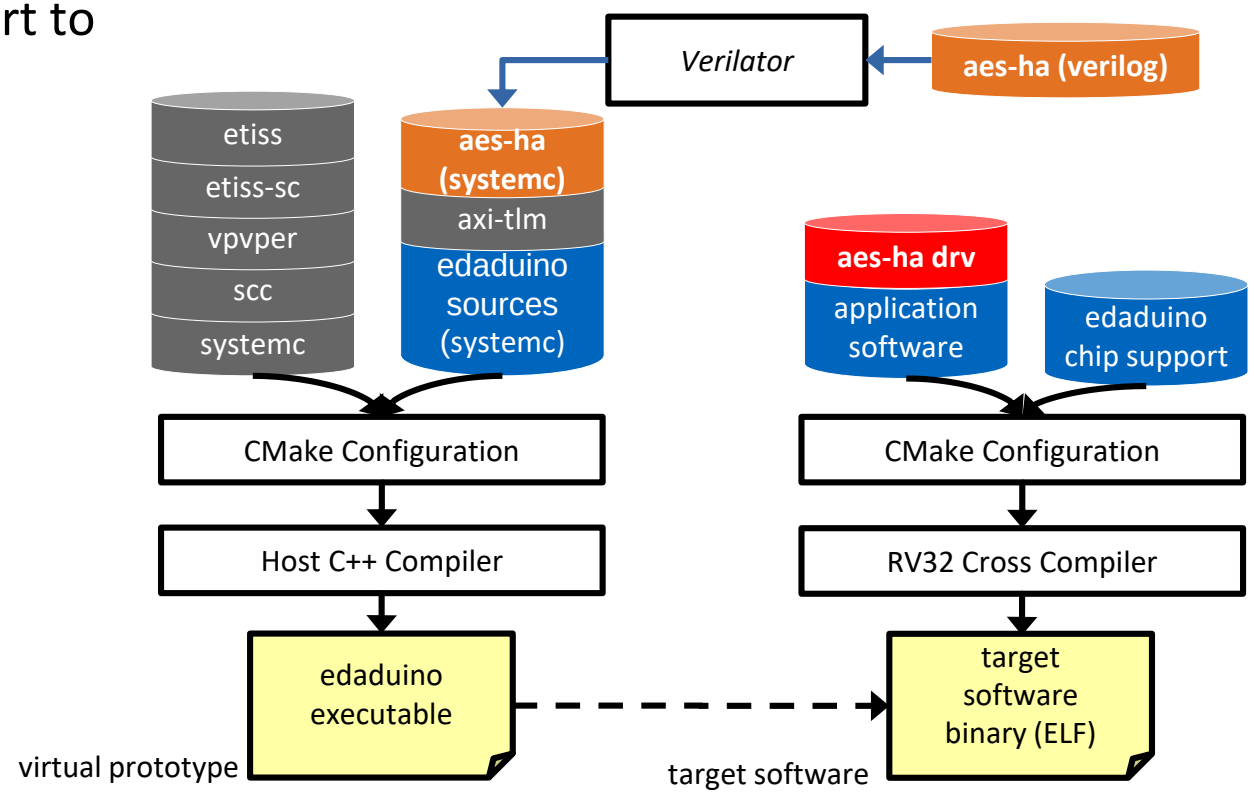
-> Tests at each design stage



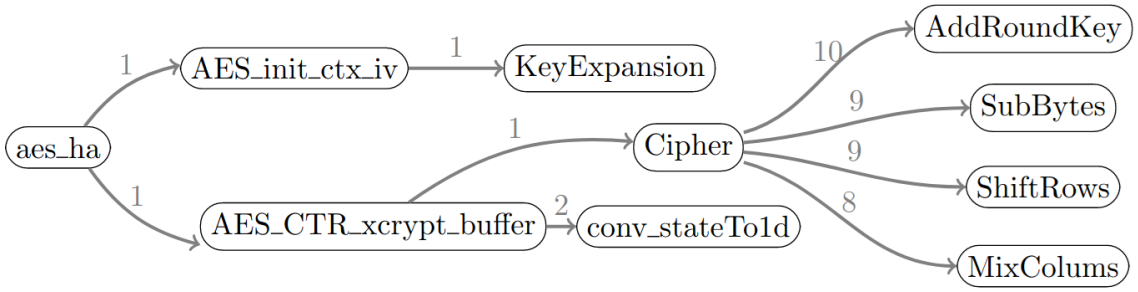
Extended Design Flow: AES-CTR to AES-HA

1) AES-CTR (AES128 w/ CTR BCMO) as C software library
2) Vivado-HLS builds RTL (normally we would export to a Xilinx FPGA design suite)
3) Verilator builds cycle-accurate SystemC models from RTL

-> Tests at each design stage



Task 2: HLS AES-SW to AES-HA



Task: Generate and optimize a hardware accelerated solution with Vivado-HLS.

Rules:

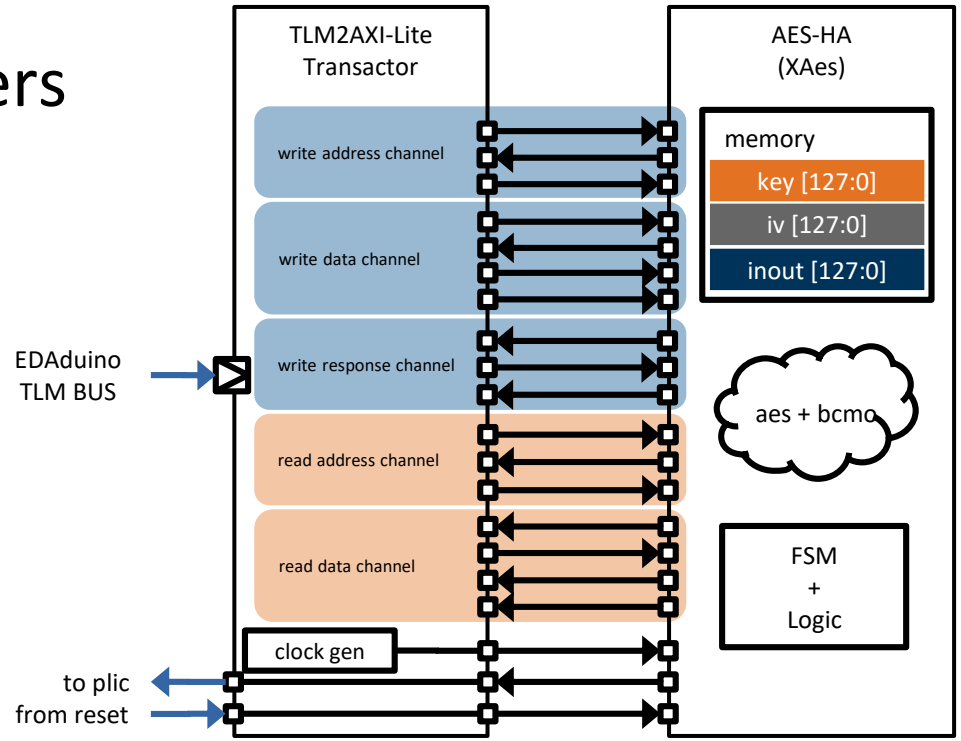
1) Do not change nor introduce additional directives to the top function aes_ha.c,
2) the design is implemented for the Xilinx FPGA chip xc7s15-ftgb196-2,
3) no resource usage shall exceed 100%,
4) a maximum latency of 500 clock cycles, and
5) the solution must pass the Vivado-HLS C/RTL Cosimulation test)

M3.3 Task 3: AES-HW Integration

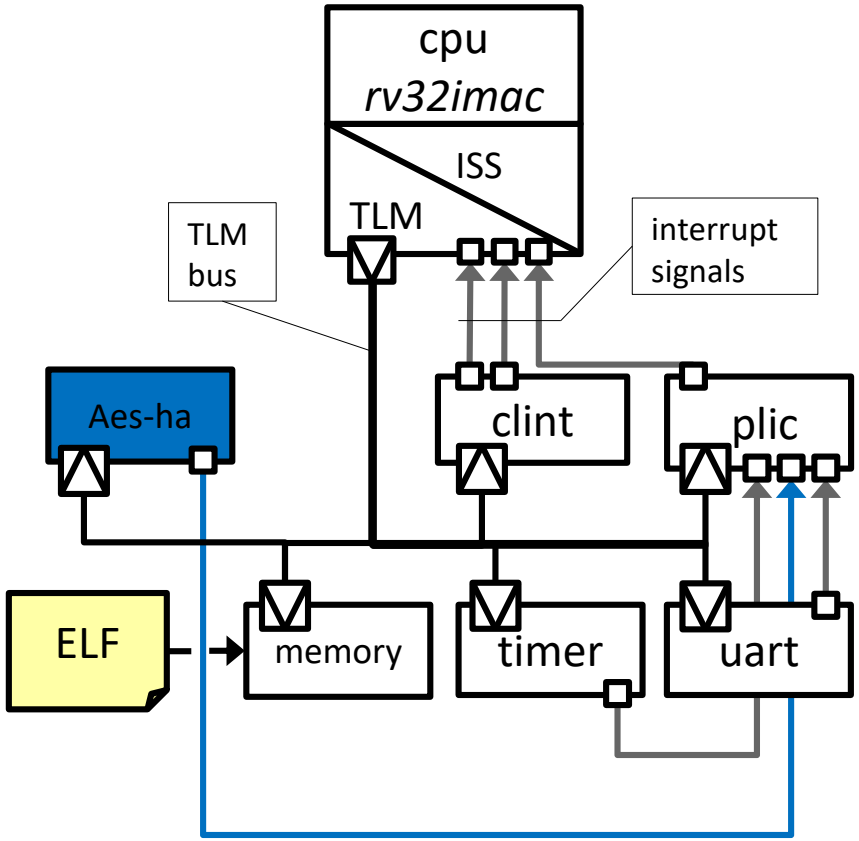
EDAduino AES-HA Integration (Basic)

Transactor provides memory-mapped registers for CPU and routes data flow AES-HA (VRTL) and registers

- Key, IV, readable and writeable by CPU
- IV updated by CTR after each AES-HA run
- Interrupt gen. by FSM signals to CPU



Task 3: HA No DMA Implementation



- No Direct Memory Access from **aes-ha**
- AXI-lite target port for control
- Same AXI-lite target port for I/O
- Driver library generated by Vivado-HLS

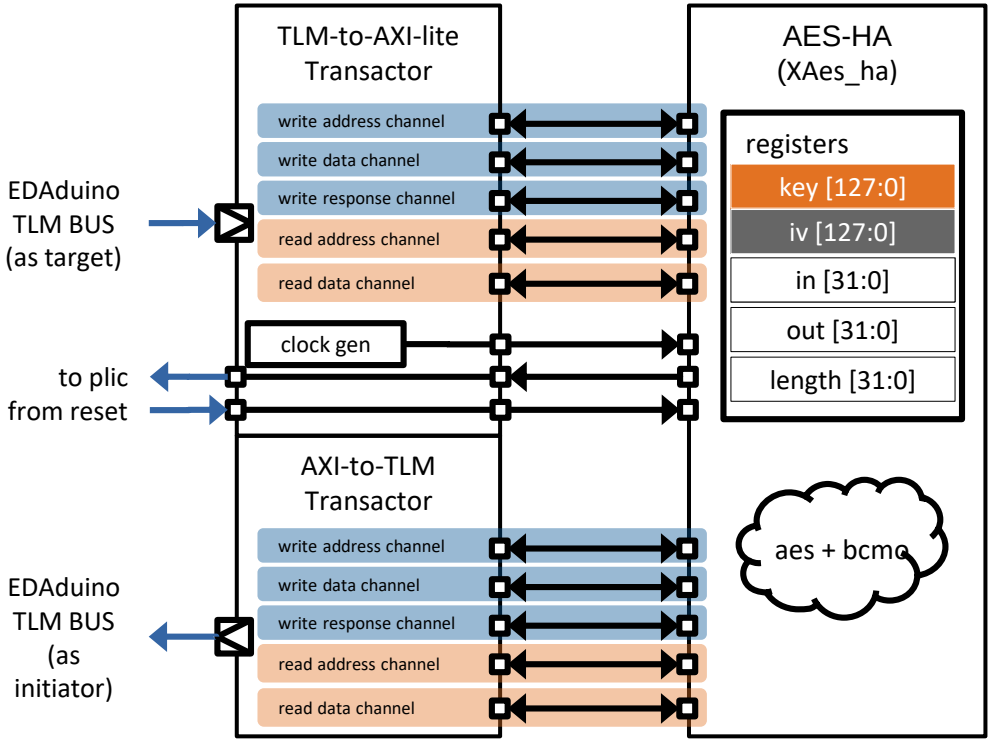
- CPU has to …
  → read message from memory
  → write input register
  → issue start command
  → synchronize (wait)
  → read output register
  → write to memory

Task: Integrate, drive and evaluate the hardware accelerated cryptography on EDAduino. **#cycles spent for this solution?**

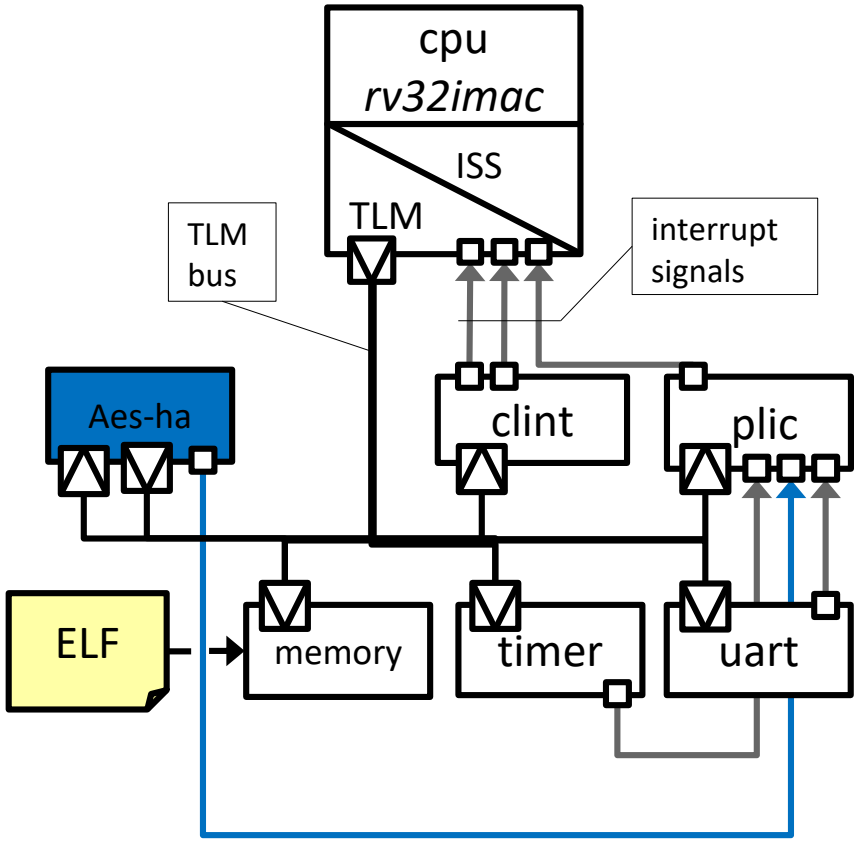
EDAduino AES-HA Integration (DMA)

AES-HA with AXI4-M (Initiator) port for direct memory access of AES-HA

- Key, IV, readable and writeable by CPU
- Message length, Input address and output address writeable
- IV updated by CTR after each AES-HA run
- Interrupt gen. by FSM signals to CPU
- No CPU memory operations for I/O



Task 3: HA with DMA Implementation



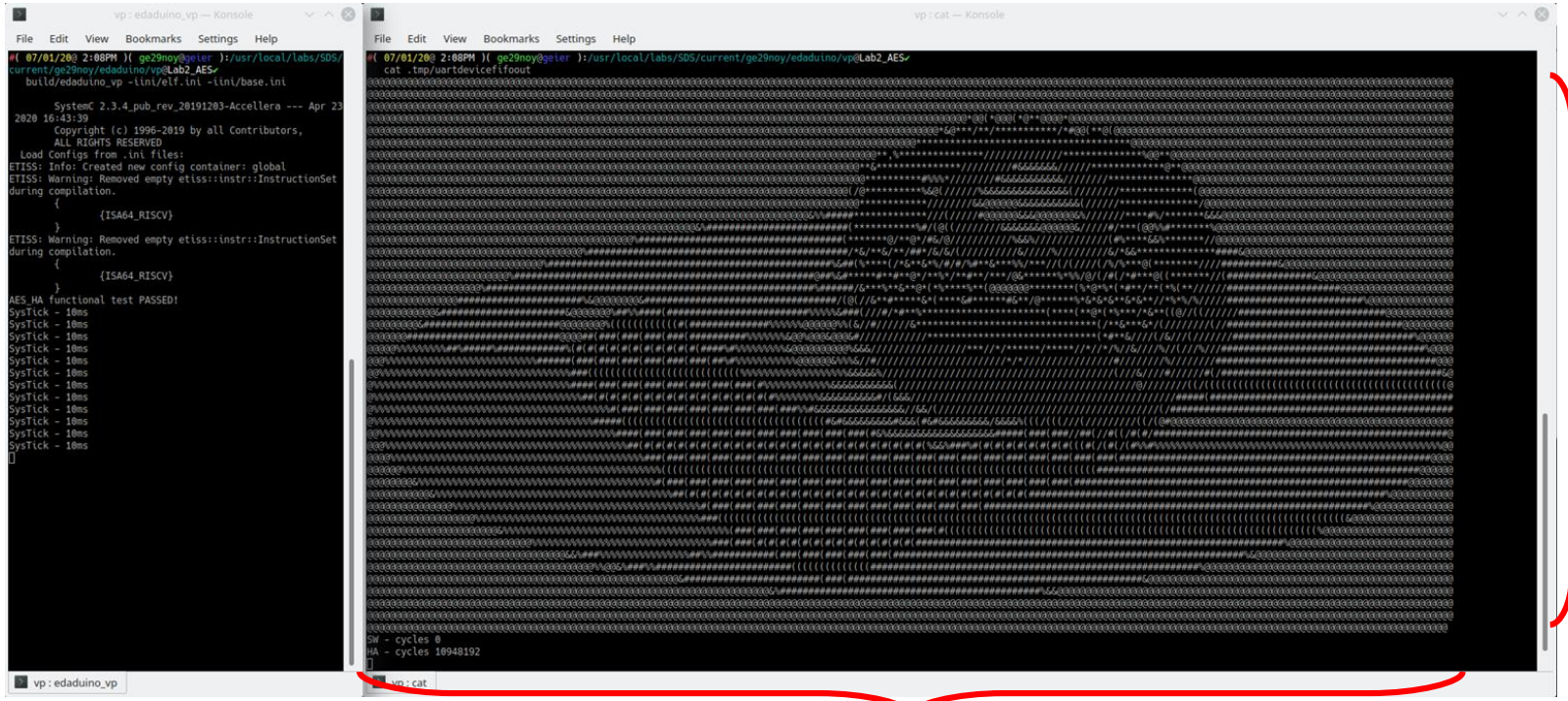
- Direct Memory Access from **aes-ha**
- AXI-lite target port for control
- AXI-M initiator port for I/O
- Driver library generated by Vivado-HLS

- CPU has to …

→ write message address to input register
→ write output destination address
→ issue start command
→ synchronize (wait)

Task: Integrate, drive and evaluate the hardware accelerated cryptography on EDAduino. **#cycles spent for this solution?**

Reference: Decrypted Message



Exam Preperation

- Solution Slides provided for the 3 modules
- Knowledge of Vivado HLS tool and most common directives
- Exam from last WS24 in HLS to practice (in a zip folder on TUWEL)

  (similar expectations in 90mins)

INFO

The registration for the second lab exam is already open!

Working on your own

Options for Accessing the Lab Material

- Lab Access:
  - TILab accounts are generated by now and should be available next week (Room 4)
  - Access with your student card (https://www.tilab.tuwien.ac.at/howto.shtml)
- Virtual Machine:
  - Using the TILab account, a virtual machine containing all sources and dependencies can be downloaded
- Source Files on gitlab:
  - A TUWEL assignment was opened asking for gitlab IDs. We will use this to give you access to the material
- Lab Manual will be uploaded to TUWEL

Warning

Warning: This lab has been reworked recently. So you might encounter Bugs. Please report them!!!

2nd Warning

Warning: The VM has been recently created and is giant (119GB uncompressed) and 52 GB compressed!

If you want to use it, test it ASAP!

Decompressing will take ages if you don’ t find a tool with xz algorithm

(USE LINUX ^^)

Thank you for your attention!