UC San Diego

# Efficient Hardware Design Using HDL & HLS

Bridging Software Expertise to Hardware Acceleration

Wuqiong Zhao

May 10, 2025

Department of ECE, UC San Diego

## Outline

## Introduction

## 🏞️ Shifting Landscape of Computation

**The End of an Era?**:

- Moore's Law slowing (transistor density).
- Dennard scaling ended (power density).
- $\Rightarrow$ Performance gains from general-purpose CPUs/GPUs are diminishing.

**The Rise of Data-Intensive Workloads**:

- AI/ML (e.g., LLMs), Big Data, IoT.
- Massive computational power and efficiency.

**The Need for Specialization**:

- CPUs/GPUs are not always optimal.
- Domain-specific architectures offer a path to continued performance and efficiency gains.

**Attempts in Industry**

New hardware targets for existing AI/ML software:

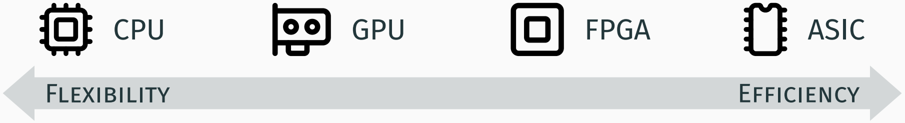
- Google **TPU**
- Google `hls4ml` ↗
- AMD **Vitis**™ ↗
- AMD **FINN** ↗
- Intel **OpenVINO**™ ↗

## What if You Could Design Your Own Chip?

**Analogy**

- Software developers write software for fixed hardware.
- Hardware designers design the hardware itself.
- Full-stack engineers can do both, in an orchestrated way!

For the task you are aiming, do you want the hardware work in a different way than the general-purpose one?



## FPGA — Field Programmable Gate Array

**Programmable Hardware Fabric**: Imagine a vast array of configurable logic blocks (CLBs) and programmable interconnects.

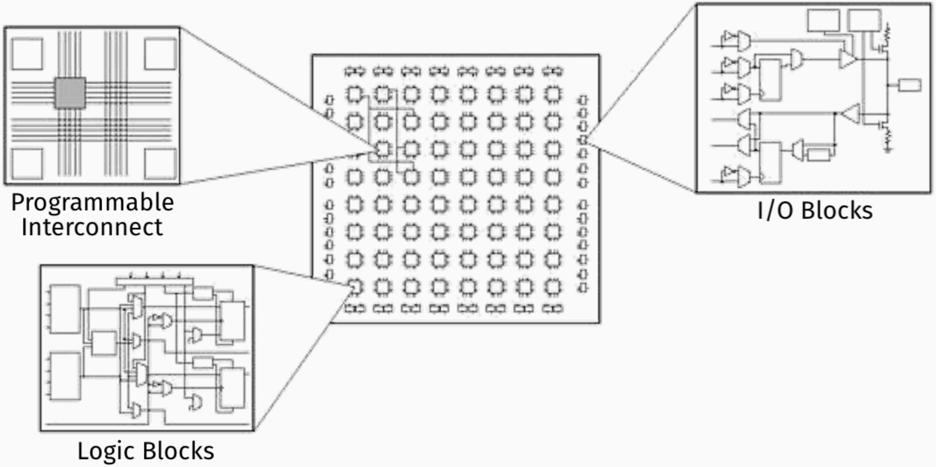
**Customization After Manufacturing**: Unlike ASICs (Application-Specific Integrated Circuits) which are fixed, FPGAs can be *reprogrammed* for different functionalities.

**Key Advantages for Custom Architectures**:

- **Tailored to Algorithm**: Design hardware directly for specific algorithms.
- **Massive Parallelism**: Exploit fine-grained parallelism inherent in algorithms.
- **Low Latency**: Data can flow through custom datapaths without OS overhead or general-purpose instruction processing.
- **Power Efficiency**: Optimize for specific operations, reducing overhead.

🎨 *"FPGAs offer a blank canvas for digital architects."*

## FPGA — Field Programmable Gate Array



The different parts of an FPGA. Adapted from `https://ni.scene7.com/is/image/ni/swvvifhq55851?scl=1`.

## FPGA — Field Programmable Gate Array

More detailed composition of an FPGA:

- **Configurable Logic Blocks (CLBs)**: Basic building blocks of FPGAs (LUTs/FFs).
- **Interconnects**: Programmable connections between CLBs.
- **I/O Blocks**: Interfaces for external communication.
- **DSP Blocks**: Specialized for high-performance arithmetic operations.
- **Memory Blocks**: Embedded memory resources.
- **Clock Management**: Resources for clock distribution and management.
- **Configuration Memory**: Stores the configuration data for the FPGA.
- **Power Management**: Resources for power distribution and management.
- **Embedded Processors**: Some FPGAs include soft or hard processors for general-purpose computing.

## Thinking About ASIC?

**ASIC** = Application-Specific Integrated Circuit.

ASICs are most suitable for high-volume production with *a fixed design*. Typical use cases include baseband signal processing, video encoding/decoding, and other fixed-function accelerators.

### FPGAs are Promising

- With the fast evolution of AI/ML algorithms, ASICs for fixed functions are outdated quickly. **Reprogramming with FPGAs** is more flexible and cost-effective;
- FPGAs are more capable with the advance of technology;
- FPGA + fixed ASIC cores (like video codec) is a good combination.

## 🧰 HDL & HLS — Tools for Digital Design

Like writing software, we do not write the assembly code directly. Instead, we can describe the hardware *gate logic* or *higher-level behavior*.

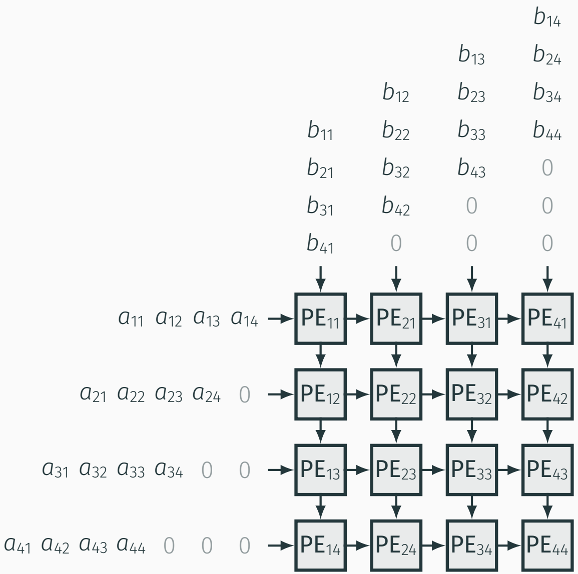
They are tools, and it is hardware architecture designs that matter.

## Hardware Architecture

## Hardware Architecture — Key to Efficiency

1. **Parallelism**: Process multiple operations simultaneously:
    - Data parallelism (vector operations, etc.);
    - Task parallelism (concurrent execution paths).
2. **Pipelining**: Overlap execution stages for higher throughput:
    - Operation pipelining (breaking complex operations into stages);
    - Strategic register insertion for improved timing and throughput.
3. **Memory Hierarchy**: Optimize data access patterns:
    - Registers, caches, local buffers, external memory;
    - Minimize expensive memory accesses.
4. **Data Flow Optimization**: Minimize data movement:
    - Local processing units near data sources;
    - Direct streaming between components.
5. **Resource Sharing**: Balance area vs. performance w/ reconfigurable modules.
6. **Specialization**: Custom datapaths for specific algorithms.

# 📝 Example: General Matrix Multiplication (GEMM) Using Systolic Array



$b_{14}$
$b_{13}$ $b_{24}$
$b_{12}$ $b_{23}$ $b_{34}$
$b_{11}$ $b_{22}$ $b_{33}$ $b_{44}$
$b_{21}$ $b_{32}$ $b_{43}$ 0
$b_{31}$ $b_{42}$ 0 0
$b_{41}$ 0 0 0
$a_{11}$ $a_{12}$ $a_{13}$ $a_{14}$
$a_{21}$ $a_{22}$ $a_{23}$ $a_{24}$ 0
$a_{31}$ $a_{32}$ $a_{33}$ $a_{34}$ 0 0
$a_{41}$ $a_{42}$ $a_{43}$ $a_{44}$ 0 0 0
$PE_{11}$ $PE_{21}$ $PE_{31}$ $PE_{41}$
$PE_{12}$ $PE_{22}$ $PE_{32}$ $PE_{42}$
$PE_{13}$ $PE_{23}$ $PE_{33}$ $PE_{43}$
$PE_{14}$ $PE_{24}$ $PE_{34}$ $PE_{44}$

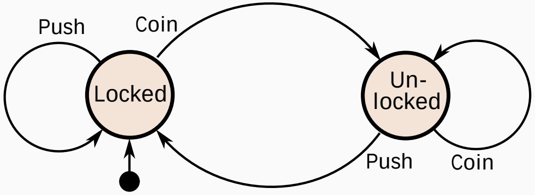
Advantages of systolic array:

- **Parallelism**: Multiple processing elements (PEs) work simultaneously.
- **Data Locality**: Data flows through the array, reducing memory access time.
- **Scalability**: Can be expanded to handle larger matrices.

For matrices with special characteristics, like symmetry, we can further optimize the systolic array.

## 🔄 Finite State Machine (FSM)

An example FSM for a coin-controlled turnstile:



For hardware design simplicity, a **2- or 3-process FSM** is recommended (personal preference). For a 2-process FSM, it involves

- **Sequential logic**: Stores the current state.
- **Combinational logic**: Determines the *(i)* next state and *(ii)* outputs based on the current state and inputs.

## Hardware Description Language (HDL)

## ❓ What is HDL and How it Works?

**Hardware Description Language (HDL)**: A specialized programming language used to describe the structure and behavior of digital circuits.

**Key HDLs**: Verilog, VHDL, SystemVerilog.

**Key Differences from Software Languages**:

- Describes *parallel* hardware structures, not sequential steps;
- Code represents physical circuit components and connections;
- Timing is explicit and critical.

**Abstraction Levels**:

- **Behavioral**: Algorithmic description;
- **Register-Transfer Level (RTL)**: Data flow between registers;
- **Gate-level**: Logic gates and connections.

## 🛠️ Basic HDL Concepts

*Examples are in Verilog.*

➕ **Combinational logic**: direct, memoryless boolean functions.

- Output depends solely on current input values;
- No memory/state - changes propagate immediately;
- Described with continuous assignments: `assign out = a & b;`.

🔁 **Sequential logic**: state-holding elements triggered by clock signals.

- Updates state values at specific clock transitions (rising/falling edges);
- Stores values in registers/flip-flops between clock cycles;
- Described in `always @ (posedge clk)` blocks.

## 🛠️ Basic HDL Concepts

🔑 Other Key HDL concepts:

- **Modules**: Encapsulation units with defined interfaces (in/out);
- **Signal assignments**:
    - Blocking (=): Sequential evaluation (seldom used in module design);
    - Non-blocking (<=): Parallel evaluation (crucial for sequential logic);
- **Simulation vs. Synthesis**: Not everything that simulates can be synthesized into hardware.

## ⛓ HDL Workflow on FPGAs

1. **HDL Design Entry**: Write RTL code in Verilog/VHDL.
2. **Functional Simulation**: Verify logic functionality.
3. **Synthesis**: Convert HDL to optimized gate-level netlist.
4. **Implementation**:
    - Translation: Map netlist to target device;
    - Placement: Position logic elements;
    - Routing: Connect logic elements.
5. **Timing Analysis**: Verify timing constraints (& locate *critical path* 🧭).
6. **Bitstream Generation**: Utilize the placement constraint file.
7. **Device Programming**: Upload bitstream to FPGA.

**For ASICs**

This is even more complicated with additional steps for fabrication!

## High–Level Synthesis (HLS)

## ⁉ What if I do not Want to Learn the HDL?

**High–level synthesis (HLS)** allows software developers to create hardware using familiar programming languages.

**Key Benefits**:

- Program in C/C++ instead of Verilog/VHDL;
- Faster development cycle (hours vs. days);
- Higher level of abstraction;
- Easier debugging and verification;
- Software-to-hardware transform.

**AMD Vitis HLS Workflow**

1. Design using C++ w/ directives;
2. **CSim**: C++ Simulation;
3. **Syn**: synthesize to RTL;
4. **CoSim**: software–hardware co-simulation;
5. **Impl**: implementation.

*Parallel Programming for FPGAs*: `hlsbook.ucsd.edu`

## ⚠️ C++ but not Exactly C++

⚔️ The conflicting nature of *sequential* software and *parallel* hardware.

**Key Differences**:

- No dynamic memory allocation;
- Limited recursion support;
- Loops often unrolled into parallel hardware;
- Function calls may be inlined as circuits;
- Limited standard library support.

**Common Pitfalls**:

- Sequential thinking leads to poor hardware;
- Ignoring variable bit-width optimization;
- Inefficient loop design creates bottlenecks;
- Non-synthesizable logics.

**Important to Remember Using HLS**

HLS tools translate direct algorithm implementations. Hardware-aware coding requires understanding architecture implications to ensure performance.

## 🎯 Finetuning HLS Design With Directives

**HLS Pragmas/Directives**: Hardware-specific annotations that guide the synthesis process without changing functional behavior.

**Common Directive Categories for Vitis HLS**:

- **Interface**: AXI, memory ports
  - `#pragma HLS INTERFACE axis port=data`
- **Loop Optimization**: unroll, pipeline, merge
  - `#pragma HLS PIPELINE II=1`
  - `#pragma HLS UNROLL factor=4`
- **Array Optimization**: partition, reshape
  - `#pragma HLS ARRAY_PARTITION variable=buffer dim=1 complete`
- **Function Inlining/Dataflow**:
  - `#pragma HLS DATAFLOW`

## 🎯 Finetuning HLS Design With Directives

### Impact on Design

- **Resource utilization** (LUT, DSP, BRAM, etc.)
- **Throughput** (initiation interval)
- **Latency** (cycle count)
- **Clock frequency** (due to critical path)

### Trade-offs

More parallelism $\Longrightarrow$ higher performance but more resources.

⚠️ *Caveat: HLS tools change frequently. Check out Vitis HLS User Guide (UG1399) for latest information.*

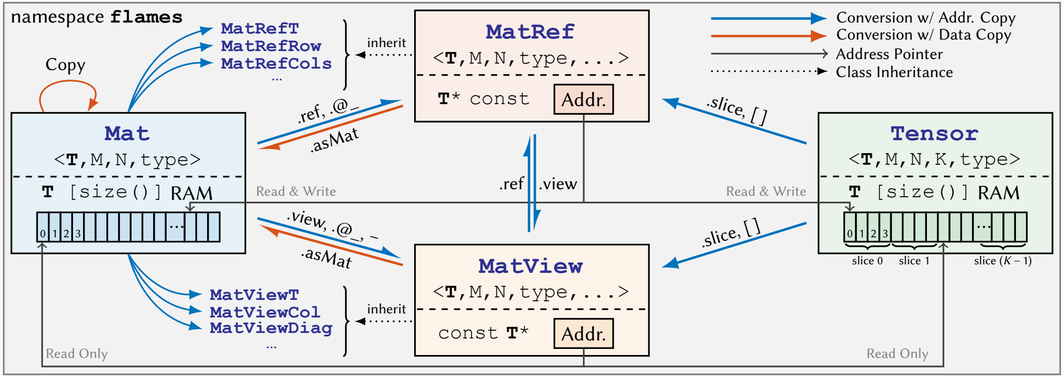
## 🔥 FLAMES HLS Library: Flexible Linear Algebra with Matrix-Empowered Synthesis

# FLAMES

High-Level Synthesis

**Vitis HLS** Support | C++14/17 | Template-Based | Header Only

# FLAMES



FLAMES high-level synthesis (HLS) library provides `class` and `template` based interface for linear algebra. [1]

## 🔥 Converting Algorithm Into FLAMES HLS Code

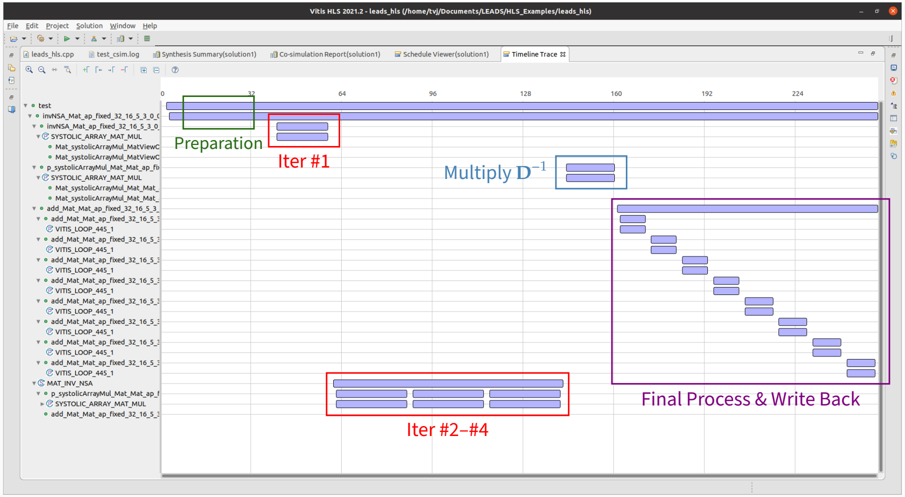
For Neumann series approximation (NSA), $\mathbf{A}^{-1} = \lim_{n\to\infty} \sum_{i=0}^{n} (-\mathbf{D}^{-1}\mathbf{E})^i \mathbf{D}^{-1}$, where $\mathbf{A} = \mathbf{D} + \mathbf{E}$, $\mathbf{D} \triangleq \mathbf{A} \circ \mathbf{I}$ is the diagonal part while $\mathbf{E}$ is the off-diagonal part. [1]

| # | Formula | FLAMES HLS C++ Implementation |
|---|---|---|
| 1 | $\mathbf{D} = \mathbf{A} \circ \mathbf{I}$ | `auto D = mat.diagMat_();` |
| 2 | $\mathbf{E} = \mathbf{A} - \mathbf{D}$ | `auto E = mat.offDiag_();` |
| 3 | $\mathbf{D}_I = \mathbf{D}^{-1}$ | `auto D_I = D.inv();` |
| 4 | $\mathbf{P} = -\mathbf{D}_I\mathbf{E}$ | `auto P = -D_I * E;` |
| 5 | $\mathbf{X} = \mathbf{P}$ (Iter. 1) | `auto X = P_ = P;` |
| 6 | for $i = 2, \ldots, n$ | `for (int i = 2; i <= n; ++i) {` |
| 7 | $\quad \mathbf{P}^i = \mathbf{P}^{i-1}\mathbf{P}$ | `    P_ *= P;` |
| 8 | $\quad \mathbf{X} = \mathbf{X} + \mathbf{P}^i$ | `    X += P_;` |
| 9 | end | `}` |
| 10 | $\mathbf{A}^{-1} = \mathbf{X}\mathbf{D}_I + \mathbf{D}_I$ | `A_inv = X * D_I + D_I;` |

Classes are templated, and there is a little C++ metaprogramming.

Overloaded functions are provided for matrix operations.

# 🔥 Timeline Trace Example



Vitis HLS 2021.2 - leads_hls (/home/tvj/Documents/LEADS/HLS_Examples/leads_hls)
File Edit Project Solution Window Help
leads_hls.cpp
test_csim.log
Synthesis Summary(solution1)
Co-simulation Report(solution1)
Schedule Viewer(solution1)
Timeline Trace
0
32
64
96
128
160
192
224
test
invNSA_Mat_ap_fixed_32_16_5_3_0_(
invNSA_Mat_ap_fixed_32_16_5_3_0_
SYSTOLIC_ARRAY_MAT_MUL
Mat_systolicArrayMul_MatViewC
Mat_systolicArrayMul_MatViewC
p_systolicArrayMul_Mat_Mat_ap_fi
SYSTOLIC_ARRAY_MAT_MUL
Mat_systolicArrayMul_Mat_Mat_
Mat_systolicArrayMul_Mat_Mat_
add_Mat_Mat_ap_fixed_32_16_5_3_
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
add_Mat_Mat_ap_fixed_32_16_5_3
VITIS_LOOP_445_1
MAT_INV_NSA
p_systolicArrayMul_Mat_Mat_ap_f
SYSTOLIC_ARRAY_MAT_MUL
add_Mat_Mat_ap_fixed_32_16_5_3
Preparation
Iter #1
Multiply $\mathbf{D}^{-1}$
Final Process & Write Back
Iter #2–#4

NSA CoSimulation Timeline for an 8 $\times$ 8 real matrix with 4 iterations.

## 🔥 More About FLAMES

Website: `flames.autohdw.com` | GitHub: `autohdw/flames` | PDF

Hardware-friendly designs: [1]

1. **Optimized RAM usage**: fixing no return value optimization (RVO) problem;
2. **Configurable parallelism**: using pragmas to control parallelism;
3. **Optimized matrix operations**: function overloading.

**— Limitations of FLAMES**

- Just a proof of concept.
- Data streaming not considered.
- Pragmas configurations are mostly limited to the global scope.

## Design Automation

## Sad Fact of Copilot Integration

Difficult to get AI tools to write correct & efficient implementations of hardware!

- AI models trained with little accessible RTL/HLS code;
- Hardware requires precise timing, resource awareness, and physical constraints (case-by-case implementation & optimization);
- AI tools struggle with synthesizable vs. simulation-only constructs.

**Current Best Practice**

Use AI for initial templates and algorithmic sketches, but rely on hardware expertise for implementation details and optimizations.

## 💻 eDSLs — Embedded Domain Specific Languages

**What are eDSLs?** Specialized languages embedded within a host language.

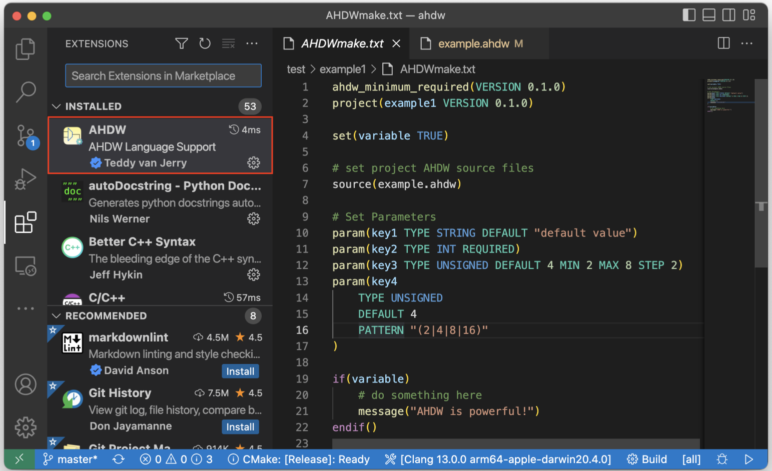
**Benefits for Hardware Design**:

- Combines host language power with hardware abstractions;
- Automated design generation and verification;
- Tight integration with software ecosystem;
- Type safety and compile-time checks.

**Notable Hardware eDSLs**

- **Chisel** (Scala-based HDL)
- **AHDW** [2] (Automatic HDW language for Verilog target)
- **PyTV/Verithon** (Python-templated Verilog)

# 💻 AHDW VS Code Extension



AHDWmake.txt — ahdw

EXTENSIONS

Search Extensions in Marketplace

INSTALLED 53

**AHDW** 4ms
AHDW Language Support
Teddy van Jerry

**autoDocstring - Python Doc...**
Generates python docstrings auto...
Nils Werner

**Better C++ Syntax**
The bleeding edge of the C++ syn...
Jeff Hykin

**C/C++** 57ms

RECOMMENDED 8

**markdownlint** 4.5M ★ 4.5
Markdown linting and style checki...
David Anson — Install

**Git History** 7.5M ★ 4.5
View git log, file history, compare b...
Don Jayamanne — Install

AHDWmake.txt | example.ahdw M

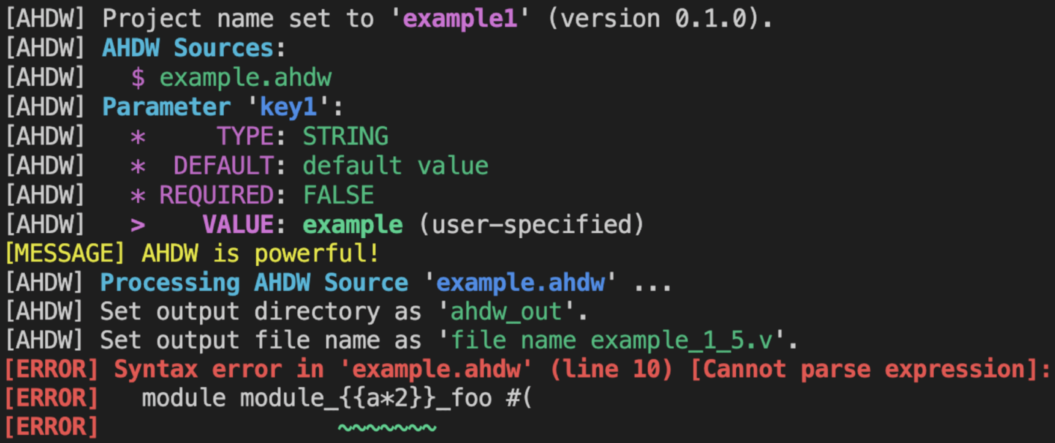
test > example1 > AHDWmake.txt

```
ahdw_minimum_required(VERSION 0.1.0)
project(example1 VERSION 0.1.0)

set(variable TRUE)

# set project AHDW source files
source(example.ahdw)

# Set Parameters
param(key1 TYPE STRING DEFAULT "default value")
param(key2 TYPE INT REQUIRED)
param(key3 TYPE UNSIGNED DEFAULT 4 MIN 2 MAX 8 STEP 2)
param(key4
    TYPE UNSIGNED
    DEFAULT 4
    PATTERN "(2|4|8|16)"
)

if(variable)
    # do something here
    message("AHDW is powerful!")
endif()
```

master* ⊗ 0 ⚠ 0 ⓘ 3 CMake: [Release]: Ready [Clang 13.0.0 arm64-apple-darwin20.4.0] Build [all]

AHDW VS Code Extension.

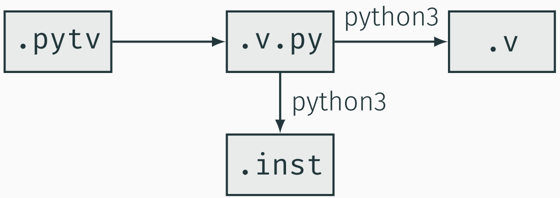
## 💻 Elegant Command Line Interface of AHDW



```
[AHDW] Project name set to 'example1' (version 0.1.0).
[AHDW] AHDW Sources:
[AHDW]   $ example.ahdw
[AHDW] Parameter 'key1':
[AHDW]   *     TYPE: STRING
[AHDW]   *  DEFAULT: default value
[AHDW]   * REQUIRED: FALSE
[AHDW]   >    VALUE: example (user-specified)
[MESSAGE] AHDW is powerful!
[AHDW] Processing AHDW Source 'example.ahdw' ...
[AHDW] Set output directory as 'ahdw_out'.
[AHDW] Set output file name as 'file name example_1_5.v'.
[ERROR] Syntax error in 'example.ahdw' (line 10) [Cannot parse expression]:
[ERROR]   module module_{{a*2}}_foo #(
[ERROR]                 ~~~~~~~
```

## 💻 PyTV — Python-Templated Verilog

GitHub: autohdw/pytv | Website: docs.rs/pytv



```
//! a = 1 + 2;              #  Python inline
assign wire_`a` = wire_b; // Verilog with variable/expression
/*!
b = a ** 2;                 #  Python block
*/
```

## Conclusion

## 🔮 Future Possibilities

1. **System Integration**:
   - Seamless CPU/GPU-FPGA heterogeneous platforms;
   - Automated hardware–software co-design frameworks;
   - System-on-chip (SoC) platforms bridging SW/HW communities;
   - Other diverse applications like SmartNIC.
2. **Hardware-Aware Software**:
   - ML frameworks with native hardware acceleration paths;
   - Domain-specific compilers optimizing for custom hardware;
   - Automated hardware-specific code transformations;
   - Cross-layer optimizations spanning SW/HW boundaries.

## 🏁 Conclusion

Key Takeaways:

1. With Moore’s Law slowing, **specialized hardware** offers a new performance frontier;
2. **FPGAs** provide a flexible middle ground between general-purpose processors and ASICs;
3. **HDLs** enable direct hardware description but require hardware thinking;
4. **HLS** bridges software and hardware domains, making acceleration more accessible;
5. **Design automation tools** like FLAMES and eDSLs further simplify hardware design.

## References

- W. Zhao, C. Li, Z. Ji, Z. Guo, X. Chen, Y. You, Y. Huang, X. You, and C. Zhang, “Flexible high-level synthesis library for linear transformations,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 7, pp. 3348–3352, Jul. 2024.
- W. Zhao, C. Li, Z. Ji, Y. You, X. You, and C. Zhang, “Automatic timing-driven top-level hardware design for digital signal processing,” in *2023 IEEE 15th International Conference on ASIC (ASICON)*, Nanjing, China, Oct. 2023.

Thanks!

PDF online: `https://go.wqzhao.org/hdl-hls-slides-25`