AI Engine Development for Versal

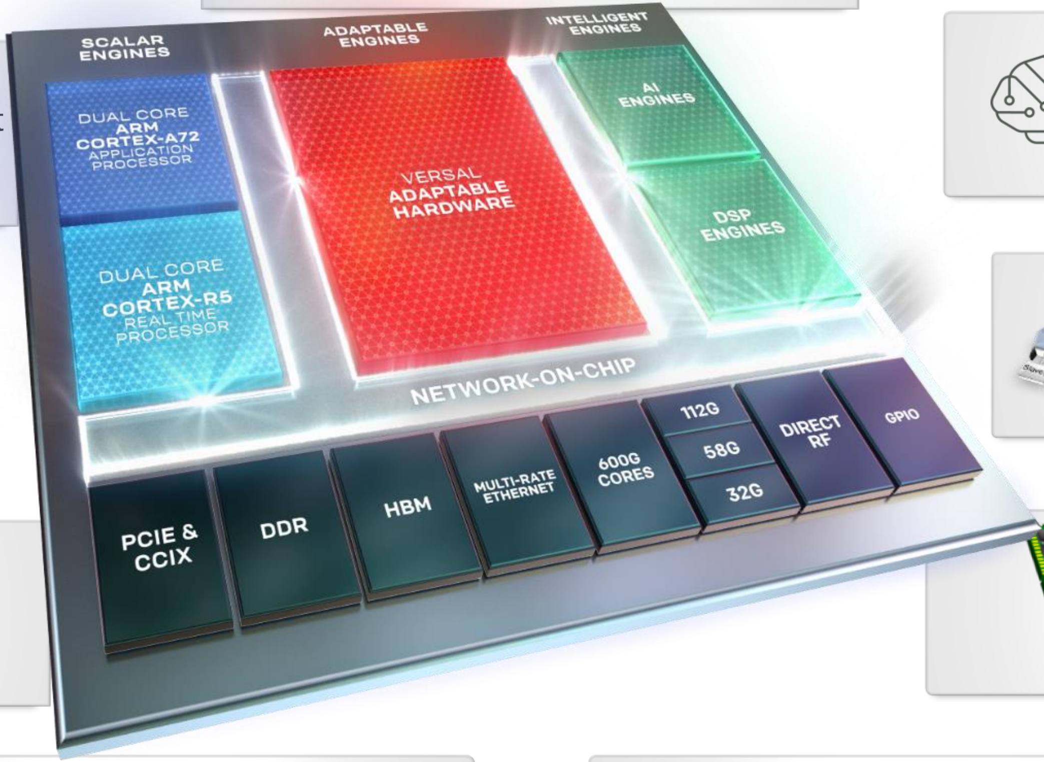
Olivier TREMOIS, PhD
SW Technical Marketing AI Engine Tools

Versal Architecture Overview

Adaptable Engines
2X compute density

Scalar Engines
- Platform Management Controller (PMC)
- Edge Compute

Intelligent Engines
- AI Compute
- Diverse DSP workloads

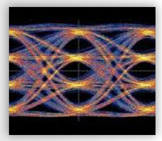


Protocol Engines
- Integrated 600G cores
- 4X encrypted bandwidth

Network-on-Chip
- Guaranteed Bandwidth
- Enables SW Programmability

Programmable I/O
- Any sensor, any interface
- Extendable peripheral set

DDR Memory
- 2X bandwidth/pin
- Server-class density

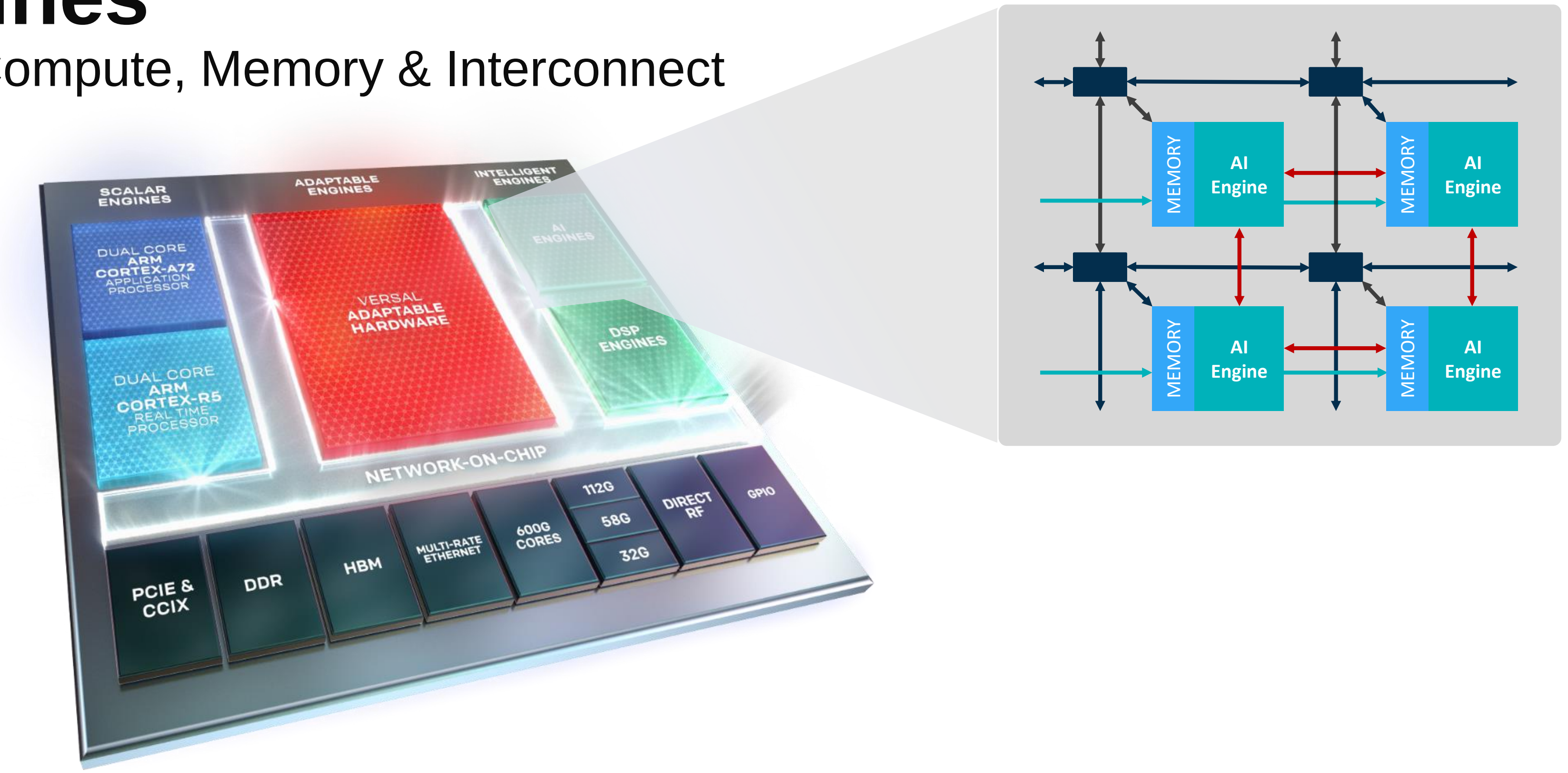


Transceivers
- Broad range, 25G →112G
- 58G in mainstream devices

PCIe & CCIX
- 2X PCIe & DMA bandwidth
- Cache-coherent interface to accelerators

AI Engines

Hardened Compute, Memory & Interconnect



Huge performance improvements versus UltraScale+

- 8x compute density @ 40% lower power

1GHz+ VLIW / SIMD vector processors

- Versatile core for ML and other advanced DSP workloads

Massive array of interconnected cores

- Instantiate multiple tiles (10s to 100s) for scalable compute

Terabytes/sec of interface bandwidth to other engines

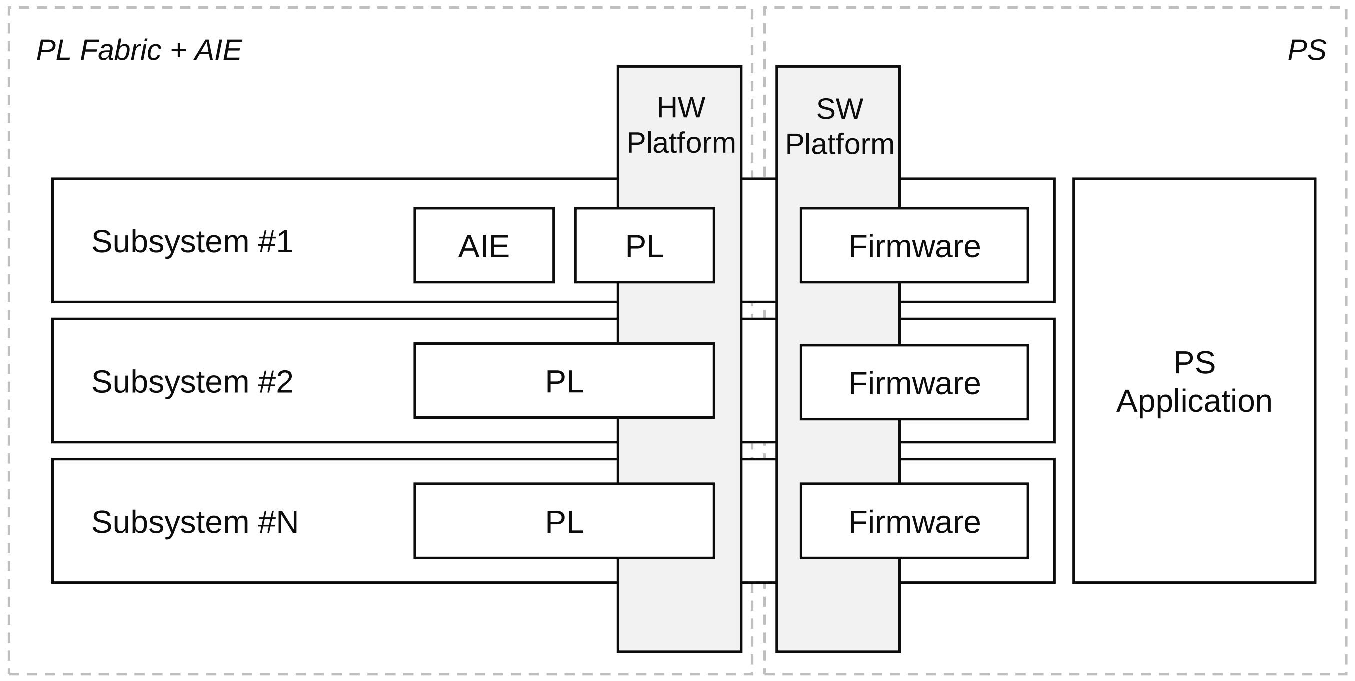
- Direct, massive throughput to adaptable HW engines
- Implement core application with AI for “Whole App Acceleration”

SW programmable for any developer

- C programmable, compile in minutes
- Library-based design for ML framework developers

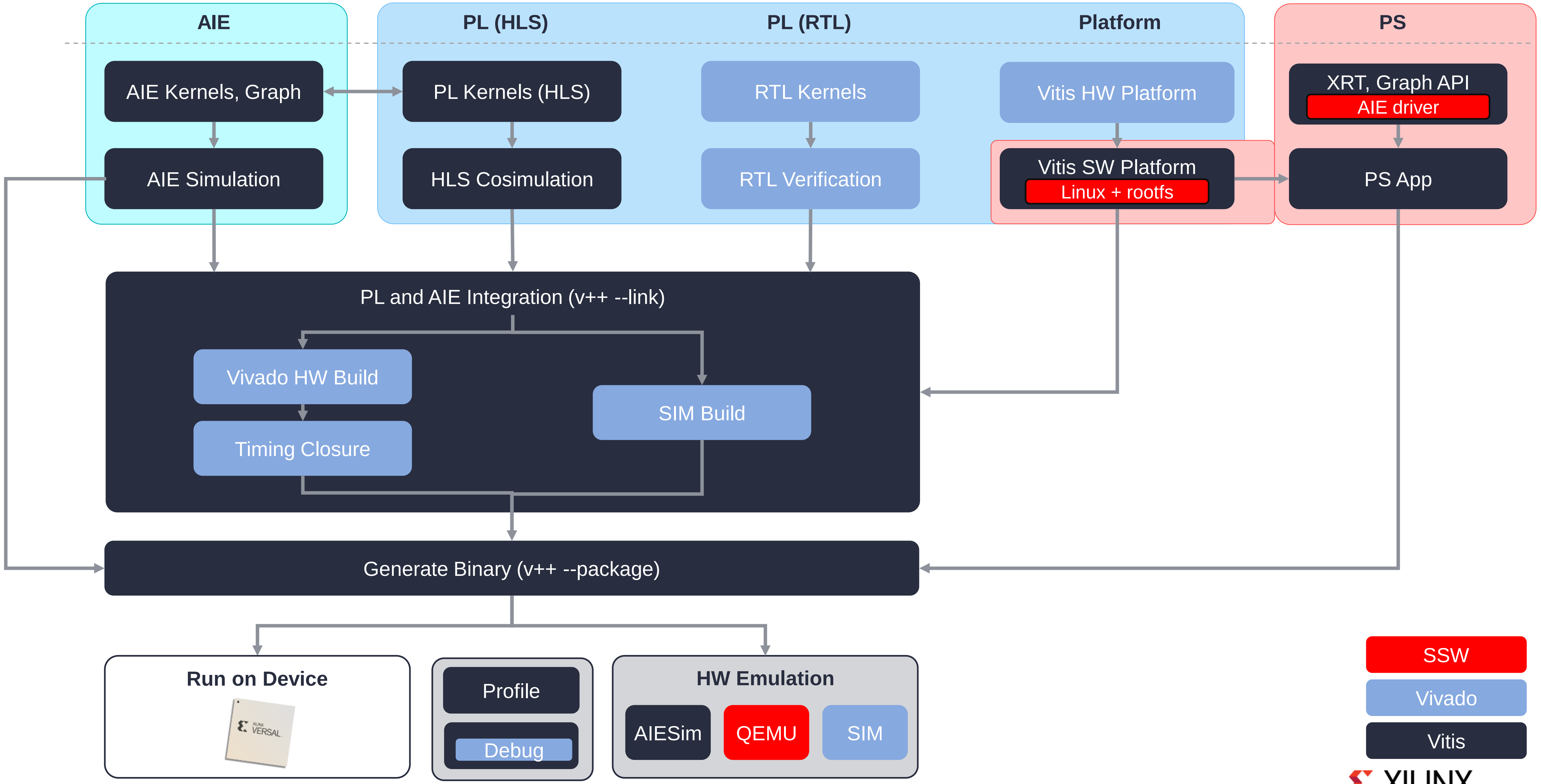
XILINX.

Vitis Philosophy : Platforms and Subsystems



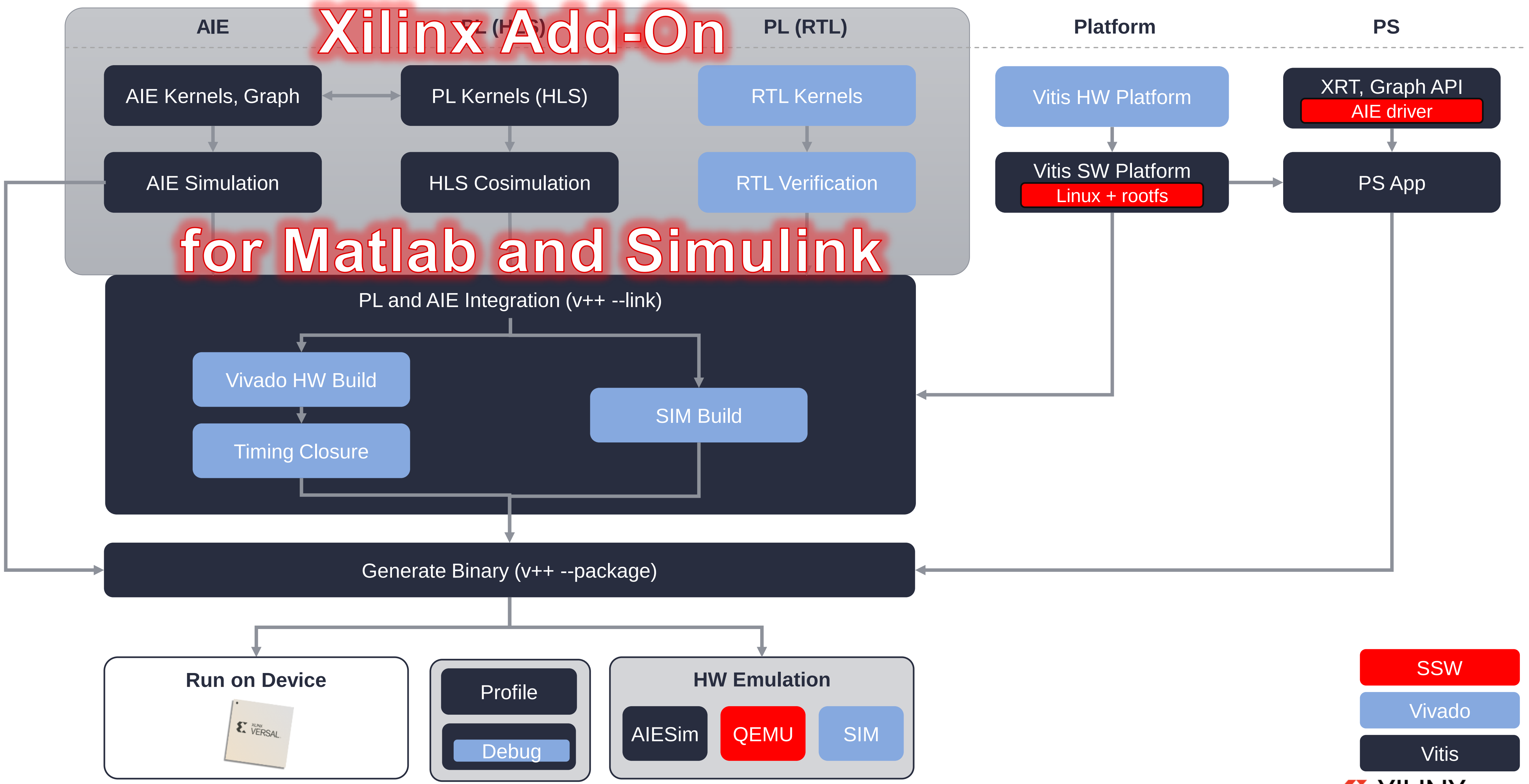
- Subsystems form the customer’s differentiating logic: AIE and PL kernels, operating under the supervision of the PS
- Versal platform provides essential infrastructure services (CIPS, NoC, I/Os, OS, Drivers…)
- Platform insulates developers from low-level details; lets them focus on application development (SW, PL or AIE)

Vitis 2020.2 Flow for Versal



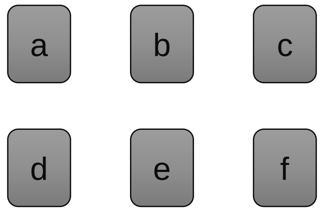
XILINX

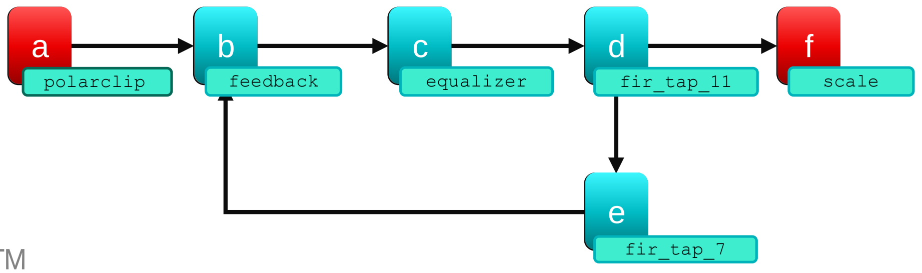
Vitis 2020.2 Flow for Versal



XILINX.

AI Engine Programming





Single Kernel Programming

> **Create AI Engine kernel programs**

> The programming model allows you to use:
>> Various Vector datatypes
>> AI Engine intrinsics
>> Window function API, …

> Analyze and Debug Kernel code
>> Compile, Simulate, profile, …

AI Engine Application

> **Create multi-kernel AI Engine projects**

> ADF graph based programming
>> Modular, hierarchical graph definition
>> Instantiation of AI Engine memories, Streams, …

> Analyze and Debug
>> Dataflow, Function scheduling, …

Programming Flow

Kernel Functional and Performance Validation

Kernel Development

Single Node Development template
Or your own single Node project
Required for profiling and low-level analysis

Kernel Validation

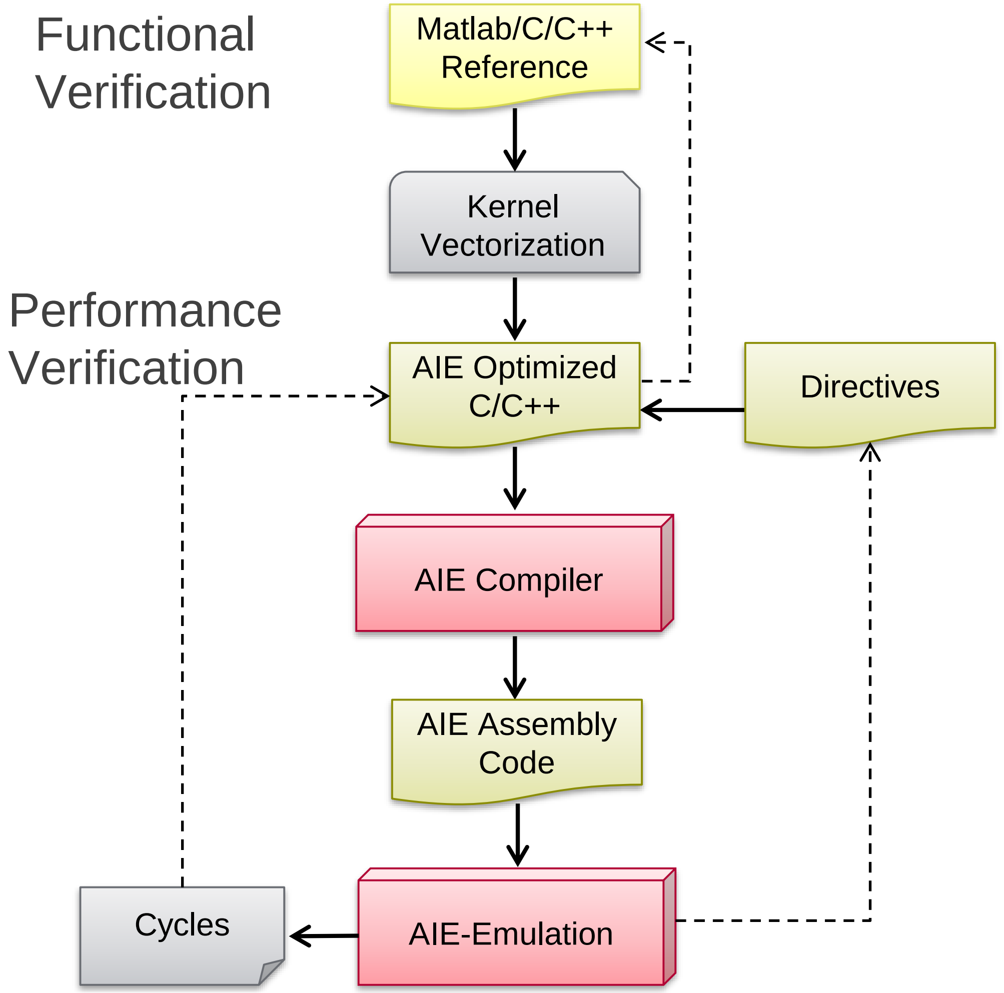
Simple connection of the graph to the environment
In-context, full AI Engine array access, PL-connection, …
Debug at the kernel level

Single Kernel Optimization

Code vectorization, vector datatypes
Vector intrinsics, optimized interface, …

XILINX

AI Engine Kernel Programming Flow



- Code restructuring
  - Vector data-types
  - Function intrinsics
  - Memory optimization
- Directives (*pragmas*)
  - Loop unrolling
  - Software pipelining
- Software development framework
  - C/C++ verification (Debugger)
  - Profiler
  - SW-Emulation: functional only
  - AIE-Emulation: cycle true

AI Engine Programming: Standard Vector Programming Techniques

Kernel Programming

```
#include <adf.h>

void fir_16taps_symm(const unsigned samples,  const int32 (&taps_in)[16],
                     input_window_cint16 * w_input,   output_window_cint16 * w_output)
{
       v16int16 coeffs;
       v32cint16 sbuff = undef_v32cint16();
       for (unsigned i = 0; i < 12 ; i++)
               coeffs = shft_elem(coeffs, (int16) taps_in[15 - i]);
       const unsigned LSIZE = (samples / 4);

       for ( unsigned i=0; i<LSIZE; i+=2)
       chess_loop_range(2,)
       chess_prepare_for_pipelining
       {
               v4cacc48 acc;
               sbuff = upd_w(sbuff, 0, window_readincr_v8(w_input));
               sbuff = upd_w(sbuff, 1, window_readincr_v8(w_input));
               sbuff = upd_w(sbuff, 2, window_read_v8(w_input)  );
               acc = mul4_sym(     sbuff , 0 , 0x3210 , 1 , 15 , coeffs, 0, 0x0000, 1 );
               acc = mac4_sym(acc, sbuff , 4 , 0x3210 , 1 , 11 , coeffs, 4, 0x0000, 1 );
               window_writeincr(w_output, srs(acc,SRS_SHIFT));

               acc = mul4_sym(     sbuff , 4 , 0x3210 , 1 , 19 , coeffs, 0, 0x0000, 1 );
               acc = mac4_sym(acc, sbuff , 8 , 0x3210 , 1 , 15 , coeffs, 4, 0x0000, 1 );
               window_writeincr(w_output, srs(acc,SRS_SHIFT));

               window_decr_v8(w_input,1);
       }
}
```

Adaptive Dataflow Library

A Kernel is a 'C/C++' function using special IO and Vector data types. It will be launched automatically by a scheduler depending on some events

Vector Datatypes for vectorized computations

Directives to help in scheduling for performance

C Window API to access data

AI Engine intrinsics to perform vectorized computation

Graph Development, Validation and optimization

Graph Development

Kernel stitching within graph

AI Engine compiler, placer and router

Can include PL-based kernel

Graph Validation

Emulation-SW: complete graph functional simulation

Emulation-AIE: Cycle true graph simulation

Debug at the graph level

Graph Optimization

I/F optimization

Location constraints, stamp (AI Engine graph map) and repeat

FIFO settings, circuit/packet switch communications, …

Graph Programming

```
#include <adf.h>
using namespace adf;

#include "kernels.h"

class myGraph : public graph {
private:
   kernel kernel1,kernel2;
public:
   input_port in;
   input_port NSamples;
   input_port Coefficients;
   output_port out;
   myGraph(){

     kernel1 = kernel::create(fir_16taps_symm);
     kernel2 = kernel::create(fir_23taps_symm);
     connect< window<128> > net0 (in, kernel1.in[2]);
     connect< window<128> > net1 (kernel1.out[0],kernel2.in[0]);
     connect< window<128> > net2 (kernel2.out[0], out);
     connect<parameter>     (NSamples, async(kernel1.in[0]));
     connect<parameter>     (Coefficients, async(kernel1.in[1]));

     source(kernel1) = "kernels/Kernel_1.cc";
     source(kernel2) = "kernels/Kernel_2.cc";
     runtime<ratio>(kernel1) = 0.1;
     runtime<ratio>(kernel2) = 0.1;
   }
};
```

Adaptive Dataflow Library

AI Engine Application described as a graph

Single Kernel Based Graph

IOs of the graph are “ports”

The constructor of the graph describes all the connections and some other parameters.

For “Single Kernel programming” this section is very simple

XILINX

Testbench

```
#include <adf.h>
using namespace adf;
```

Adaptive Dataflow Library

```
#include "kernels.h"
#include "kernels/include.h"
#include "project.h"

kernelOptGraph mygraph;
```

AI Engine Graph

```
simulation::platform<1,1> platform("data/input.txt", "data/output.txt");
connect<> net0(platform.src[0], mygraph.in);
connect<> net1(mygraph.out, platform.sink[0]);
```

Creation of a virtual platform:
- Input test vector file
- Output vector file
- Connection of the graph

```
int main(void) {
  int32 taps[16] = {-100, 200, -300, 400, -500, 600, -700, 800, 800, -700, 600, -500,
400, -300, 200, -100};

  mygraph.init();
  mygraph.run(4);
  mygraph.update(mygraph.samples, uint32(INPUT_SAMPLES));
  mygraph.update(mygraph.coefficients, taps, 16);
  mygraph.end();

  return 0;
}
```

Simulation control

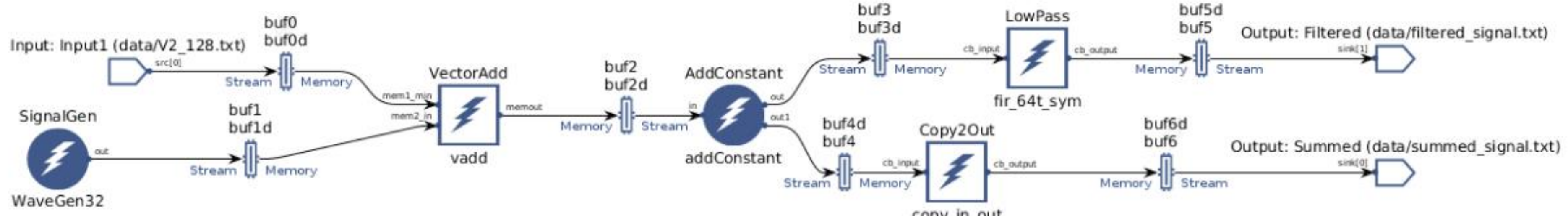
Vitis Analyzer

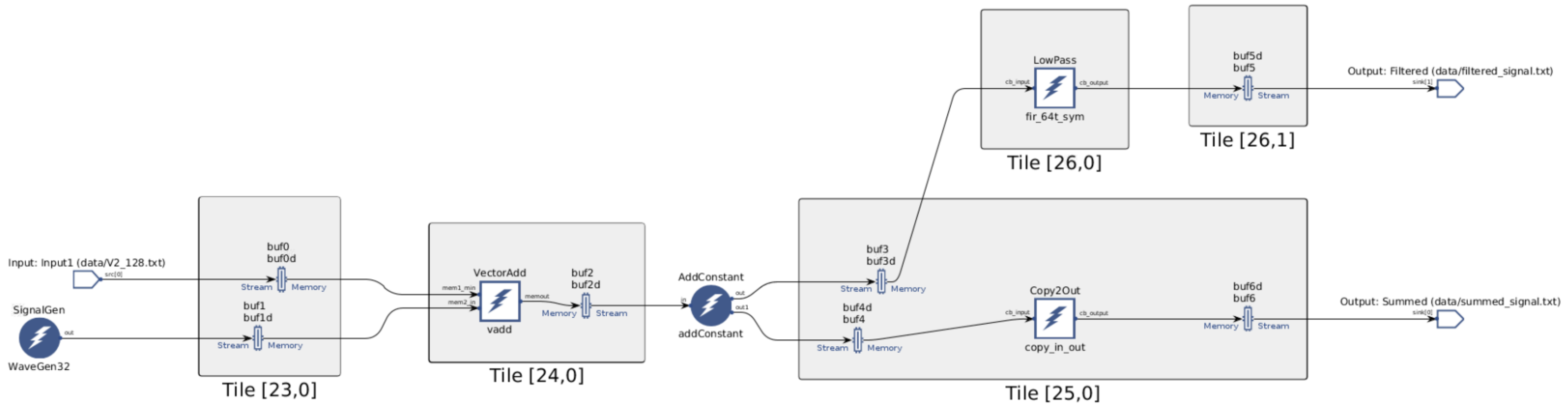
Vitis Analyzer introduction

- Compile Results Analysis:
  - Graph
  - Mapping
  - Memory footprint
  - DMAs, Locks, …
- Profiling Viewer
- Simulation Timeline analysis
- Can be used also within Makefile flow

Vitis Analyzer Compilation View

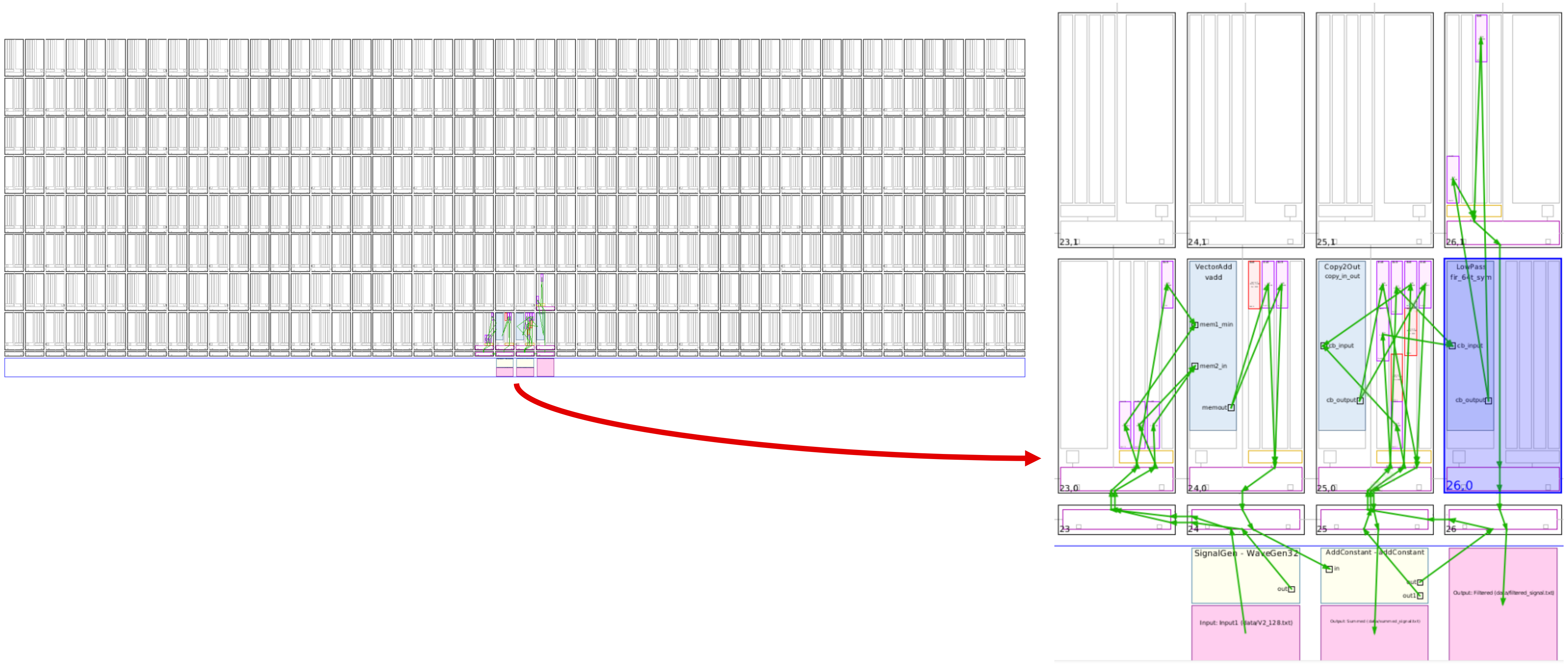
- Graph View
  - Shows all the kernels defined in the AI Engine graph (AI Engine Array and PL)
  - The kernels can be grouped by Tile or Subgraph or no grouping at all





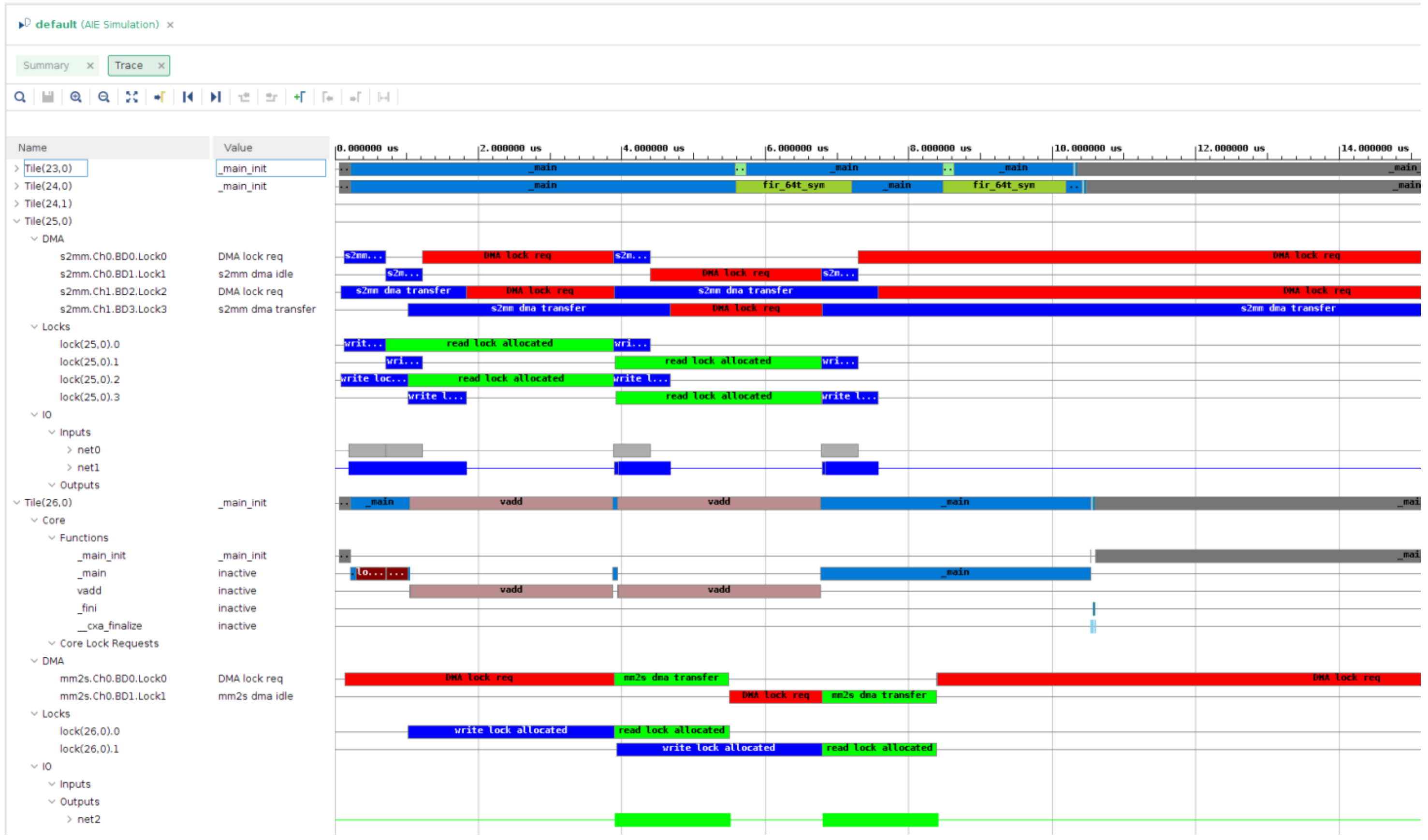
Vitis Analyzer Compilation View

- Array View
  - Shows the complete AI Engine array and specifies which Tile is used and wall connections



Vitis Analyzer Trace view

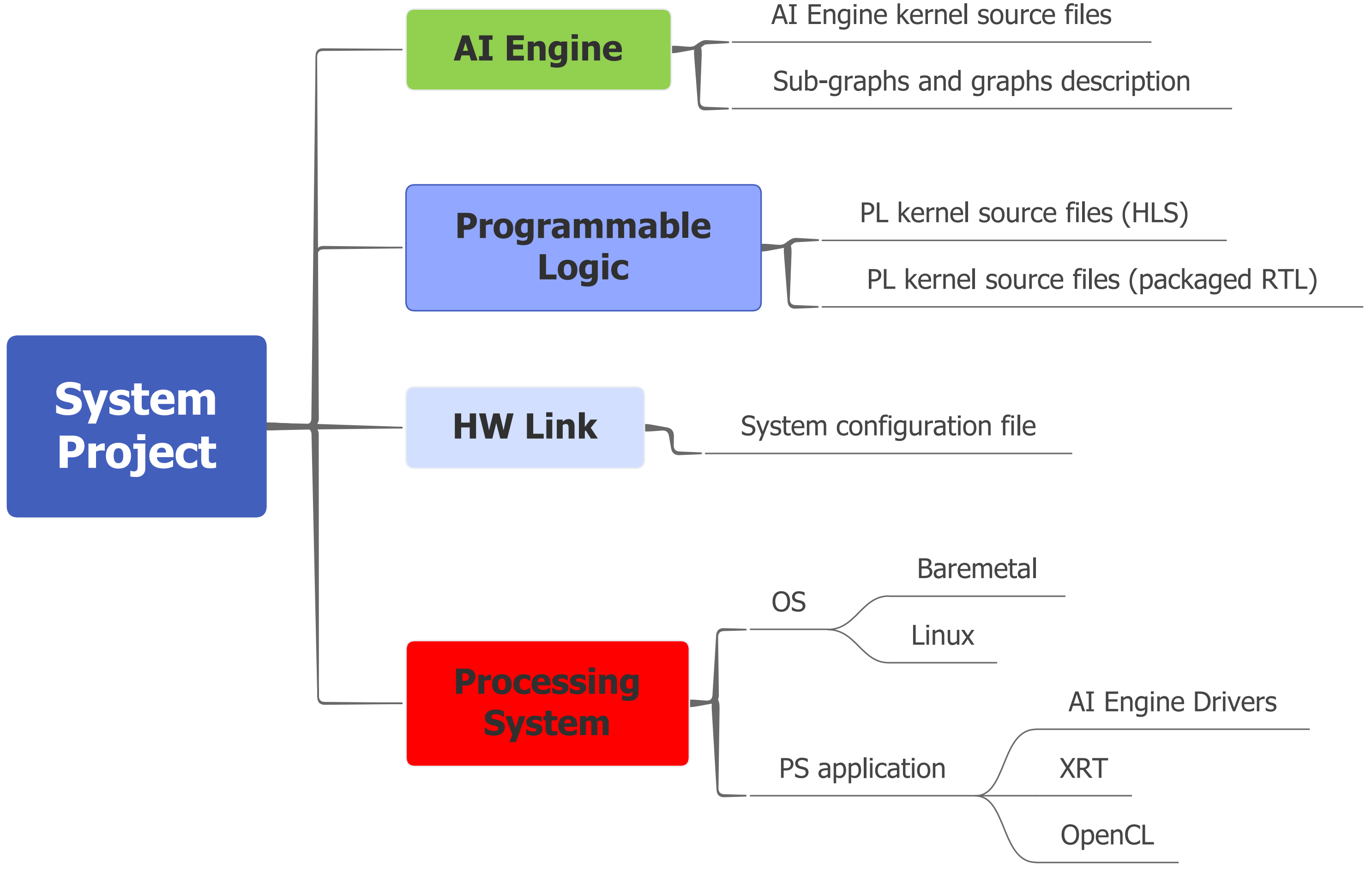
- The Trace view gives information on what runs on each tile (active tiles only) of the array:
  - Core, DMA, Locks and IOs
- A Tile is active as soon as its AI Engine processor, its local memory or its interconnect is active



Name	Value
Tile(23,0)	_main_init
Tile(24,0)	_main_init
Tile(24,1)	
Tile(25,0)	
DMA	
s2mm.Ch0.BD0.Lock0	DMA lock req
s2mm.Ch0.BD1.Lock1	s2mm dma idle
s2mm.Ch1.BD2.Lock2	DMA lock req
s2mm.Ch1.BD3.Lock3	s2mm dma transfer
Locks	
lock(25,0).0	
lock(25,0).1	
lock(25,0).2	
lock(25,0).3	
IO	
Inputs	
net0	
net1	
Outputs	
Tile(26,0)	_main_init
Core	
Functions	
_main_init	_main_init
_main	inactive
vadd	inactive
_fini	inactive
_cxa_finalize	inactive
Core Lock Requests	
DMA	
mm2s.Ch0.BD0.Lock0	DMA lock req
mm2s.Ch0.BD1.Lock1	mm2s dma idle
Locks	
lock(26,0).0	
lock(26,0).1	
IO	
Inputs	
Outputs	
net2	

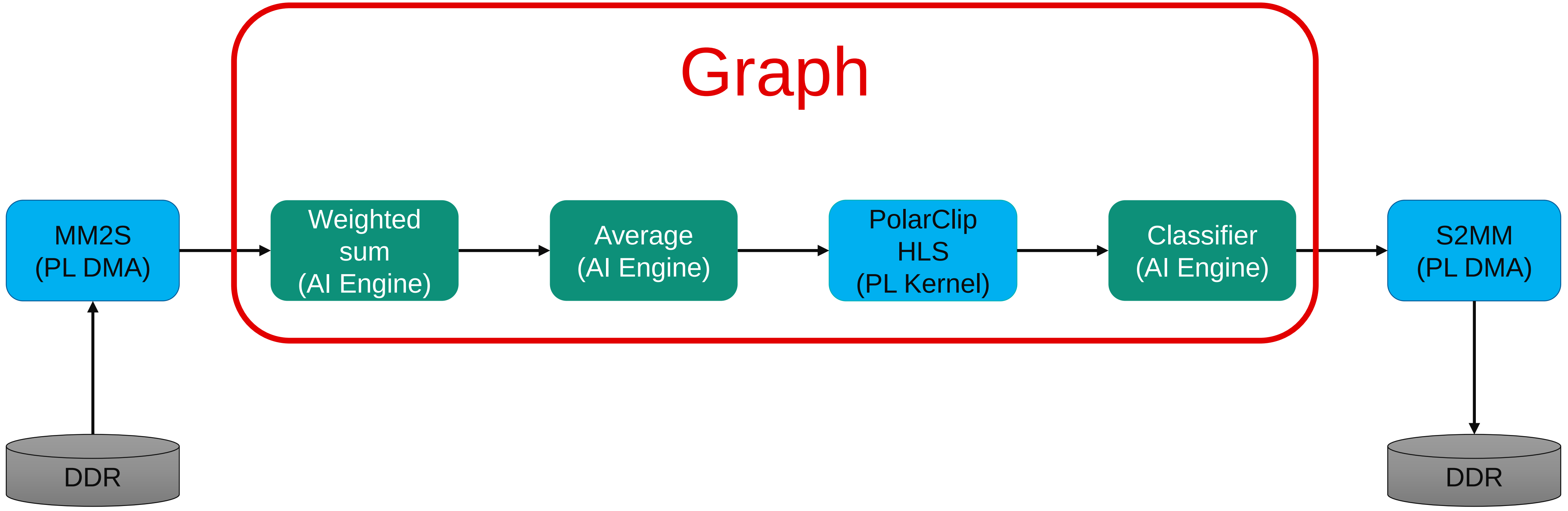
AI Engine Project Creation in Vitis 2020.2

System Project structure in Vitis

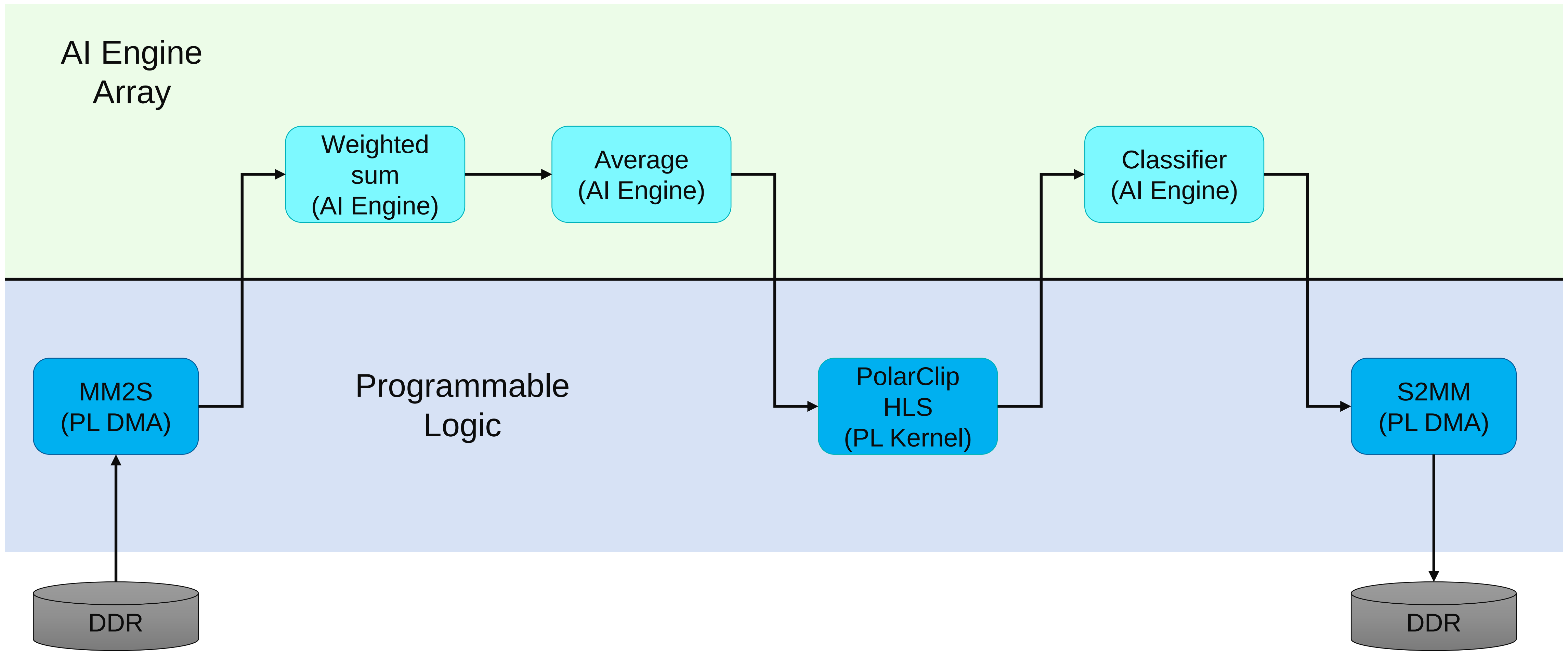

System Project
AI Engine
AI Engine kernel source files
Sub-graphs and graphs description
Programmable Logic
PL kernel source files (HLS)
PL kernel source files (packaged RTL)
HW Link
System configuration file
Processing System
OS
Baremetal
Linux
PS application
AI Engine Drivers
XRT
OpenCL

Vitis 2020.2 Demo

Example design partitioning



Example design partitioning



Vitis 2020.2
Project Creation and AI Engine Simulation

Vitis 2020.2
PL kernel compilation and HW link

Vitis 2020.2
PS app compilation and HW Emulation

Vitis 2020.2
HW Implementation

Summary

- Vitis is a unified tool that is used throughout the AI Engine development flow

- AI Engine development is a 2-stage process
  - Single kernel
  - Graph development

- Vitis handles all Versal ACAP domains

Thank You