Unit 2
Memory and Basic Processor Interfaces

EL-GY 9463: INTRODUCTION TO HARDWARE DESIGN

PROFS. SUNDEEP RANGAN, SIDDHARTH GARG

Learning Objectives

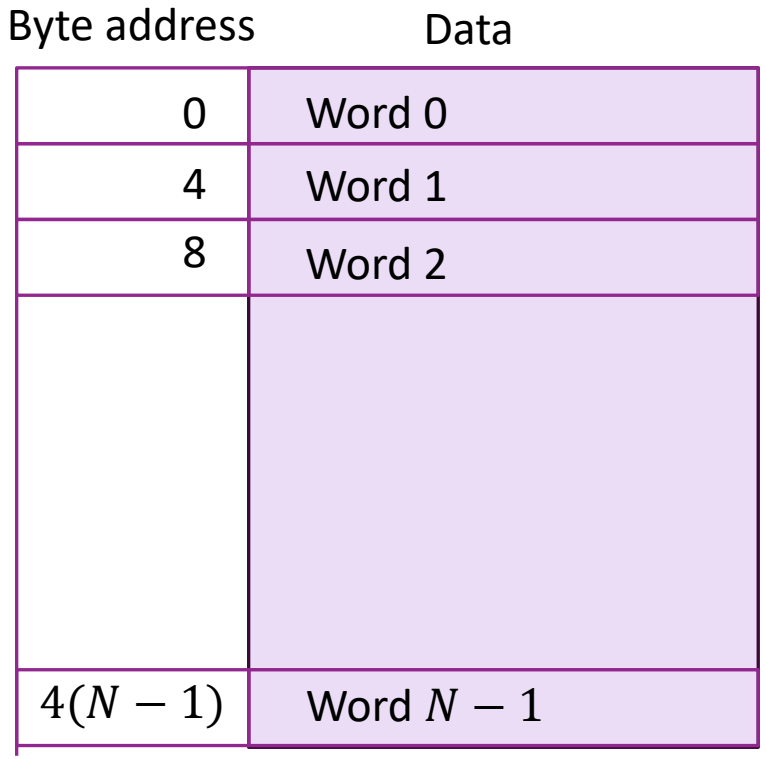
- ❑ Describe the basic components of random access memory
  - Address and data bus, address decoder, storage
- ❑ Identify which types of memory should be used for storage
  - External vs. internal
  - Registers vs. block memory
- ❑ Describe and analyze timing for the AXI4-Lite protocol for reading and writing registers
- ❑ Implement a simple IP with AXI4-Lite control registers using Vitis HLS
- ❑ Synthesize and simulate the IP and analyze the timing diagrams
- ❑ Deploy the IP on an FPGA board and interface with the IP via PYNQ

Outline

- Random Access Memory and Memory-Mapped Registers
- Processor Interfaces with AXI4-Lite
- Implementing AXI4-Lite Interface in Vitis HLS

Addressable Memory

- An array of elements in storage
  - Use for vectors, matrices, packets, data, …
- Word: Each of the data element
  - Typically, has word / data width = 32 or 64 bits
- Memory depth = number of words
- Memory size=depth × word width
- Address: A location of a word
  - Often byte-addressable
  - Word k is at byte address $4k$ when word width=32



Example 32-bit memory with N depth

Sequential vs Random Access

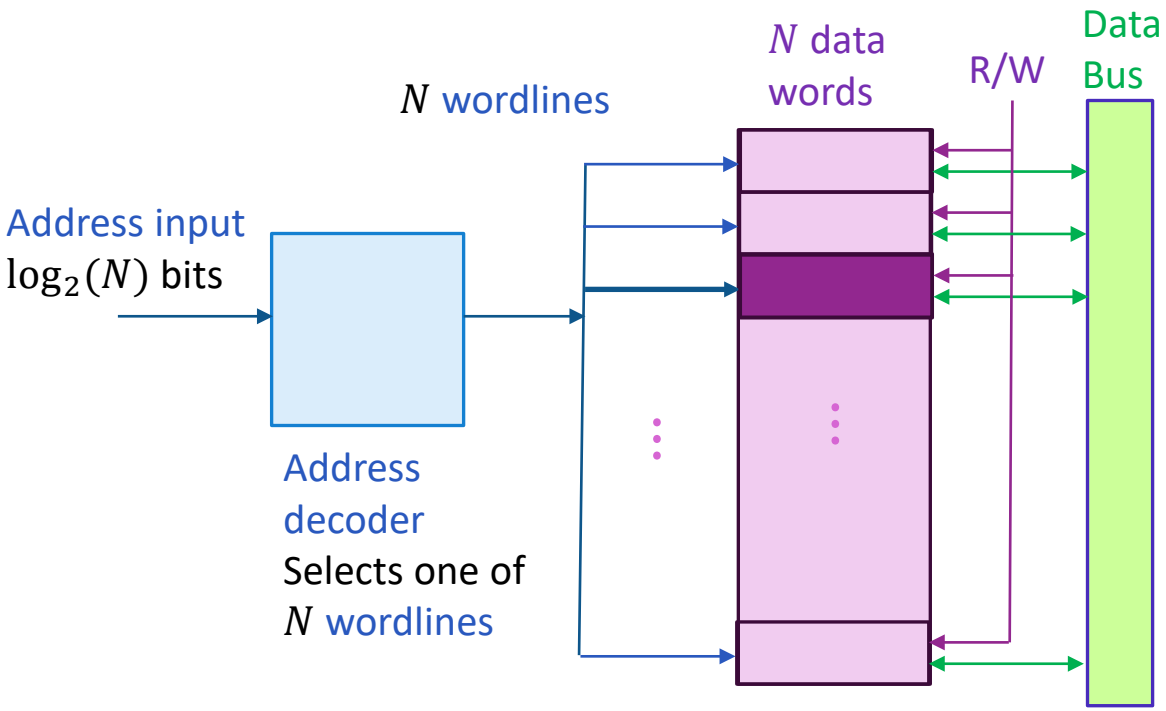
❑Sequential Access

- Access time depends on *position*
- You must move through data in order
- Ex: Hard disks (seek + rotation)
- FIFO queues (logical sequential access)

❑Random access

- Any element can be accessed in constant time
- SRAM (on chip block memory, FPGA BRAM)
- DRAM (external memory)
- Register files (architecturally random access)

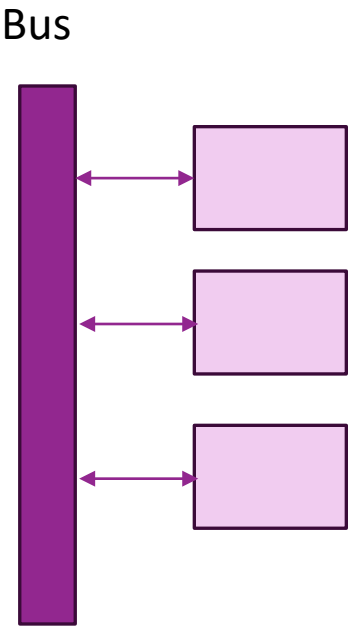
Typical RAM Architecture



- Three interface signals:
  - Address input: Which word to select
  - Read/Write: Which direction of transfer
  - Data: Value to read or write
- Address input:
  - $\log_2(N)$ bits to select one of N words
  - Each word has a wordline select
  - Selection determined by address decoder

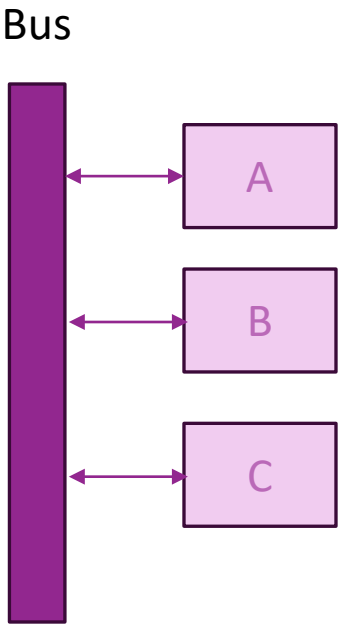
Bus

- Bus: A shared communication pathway for multiple units
  - Physically a set of wires
  - For data, addresses, and control
  - A shared protocol so only unit drives the bus at a time
- Many examples:
  - Multiple memory units connected to a common data bus
  - A processor and multiple memory units sharing address or data
  - Multiple peripherals on an SoC sharing an AXI bus
  - Multiple devices on an I^2C or SPI bus
  - …



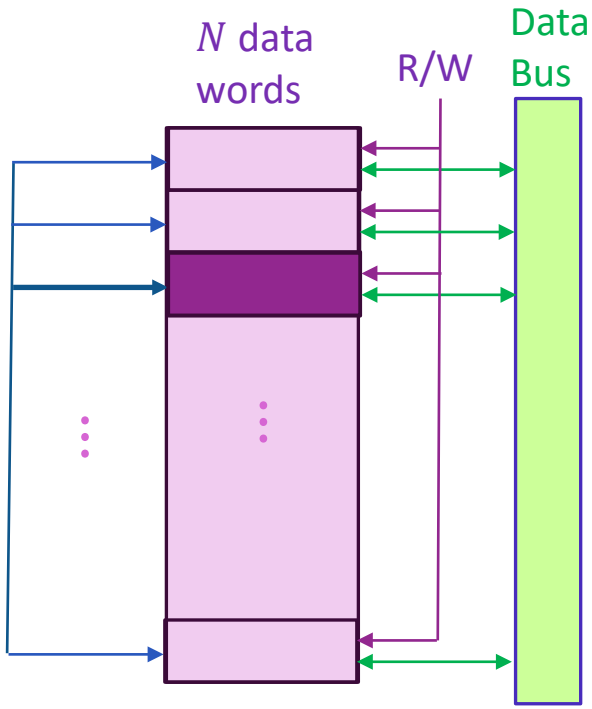
High Impedance

- Method to share a common bus
  - One unit drives a value V on the bus
  - Other units go into a high-impedance state, noted Z
  - High impedance units are disconnected
  - Bus value= V
- Sometimes called Tri-State
  - Each bit is 0, 1 or Z



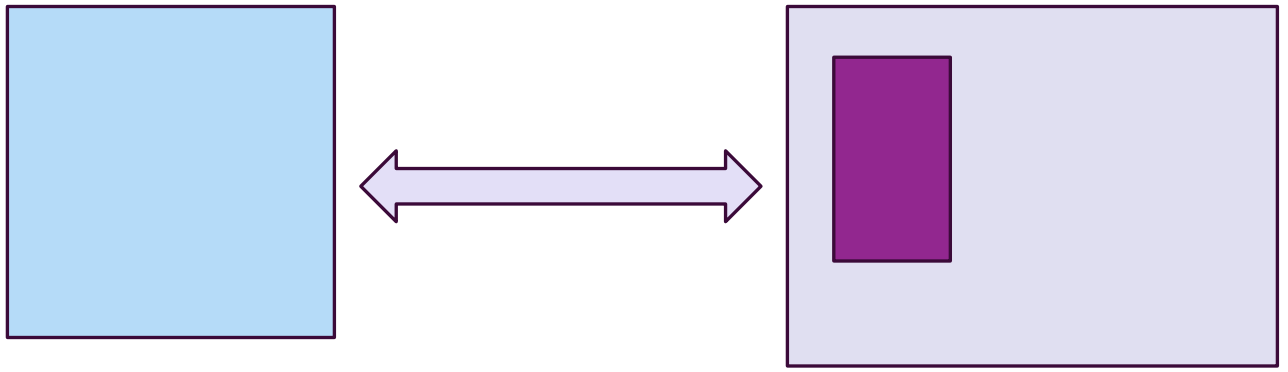
A	B	C	Bus Value	Remarks
Z	15	Z	15	B drives bus
41	Z	Z	41	A drives bus
Z	Z	Z	X	No driver- no reliable value
32	78	Z	X	Collision of A, B – no reliable value Protocols will avoid this condition

High Impedance on a Data Bus



- Memory read (Tri-State Bus)
  - Selected memory element drives its value
  - Other units go to Z
  - Data bus value = selected memory element's value
- Memory write (Broadcast + Selective Capture)
  - Data bus value appears at all units
  - Selected element loads value into its storage element
  - Other elements ignore value

External vs. Internal Memory



Processing Hardware

External memory

- Off-chip
- High capacity (GBs)
- But need I/O to process

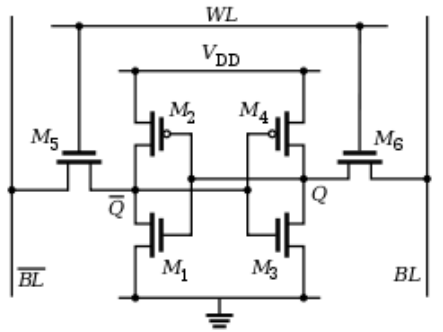
Internal memory

- On-chip
- Low capacity (KB-MB)
- Direct access to processing
- Low latency

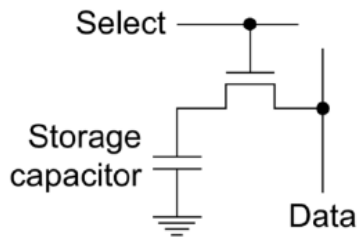
External vs. Internal Compared

Feature	Internal Memory (SRAM)	External Memory (DRAM)
Location	On-chip	Off-chip
Storage	Flip-flop / 6-T cell	Capacitor + transistor
Density	Low	High
Capacity	Small (KB–MB)	Large (GB)
Speed	Very fast	Slower
Power	Higher static	Lower static, higher dynamic
Access	Direct by logic	Through memory controller / I/O
Refresh	No	Yes

SRAM cell



DRAM cell



Example Memory Internal Sizes

Platform	On-Chip Memory Type	Typical Size	Notes
Laptop CPU (Intel Core)	SRAM (L1/L2/L3)	20–40 MB	L3 dominates; very fast
NVIDIA A100 GPU	SRAM (L2 cache)	~40 MB	HBM2e (40–80 GB) is off-chip/on-package
Xilinx Virtex-7 FPGA	BRAM (36 Kb blocks)	2–6 MB	Distributed across fabric
Custom ASIC (5 mm^2 @16nm)	SRAM	5–10 MB	Depends on layout & redundancy

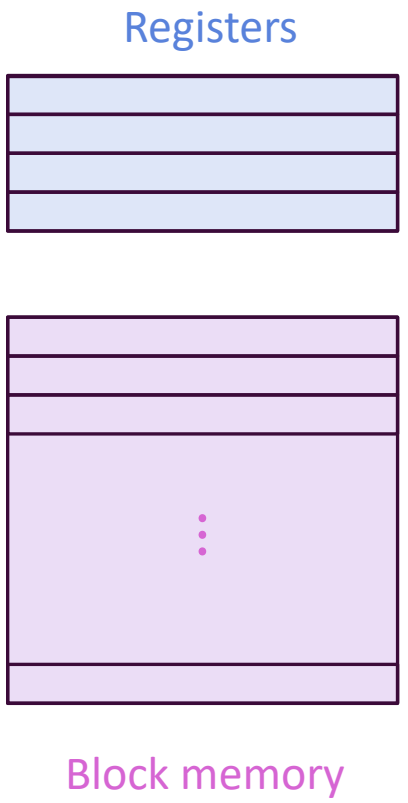
Lesson: Internal memory is limited! Use judiciously.

Example: Video

- A video processor needs to store 8 frames of raw video
  - 512×512 pixels, 3 color channels, 8-bits per pixel
- What is the memory capacity needed?
  - $B = 512 \times 512$ pixels $\times\ 3$ channels $\times\ 8$ frames $\times\ 1$ B/pixel $\approx$6.2 MB
  - A sizeable portion of memory even on a large chip
- If the data is stored as 32-bit words, how many words are in the storage
  - There are 4B/word $\Rightarrow N = \frac{6.2}{4} \approx 1.6$ Mwords
- How many address bits are needed assuming byte addressing?
  - For byte addressing we need $\mathrm{K} > \log_2(B) = 9 + 9 + 3 + \log_2(3)$
  - Smallest integer $K = 25$ bits

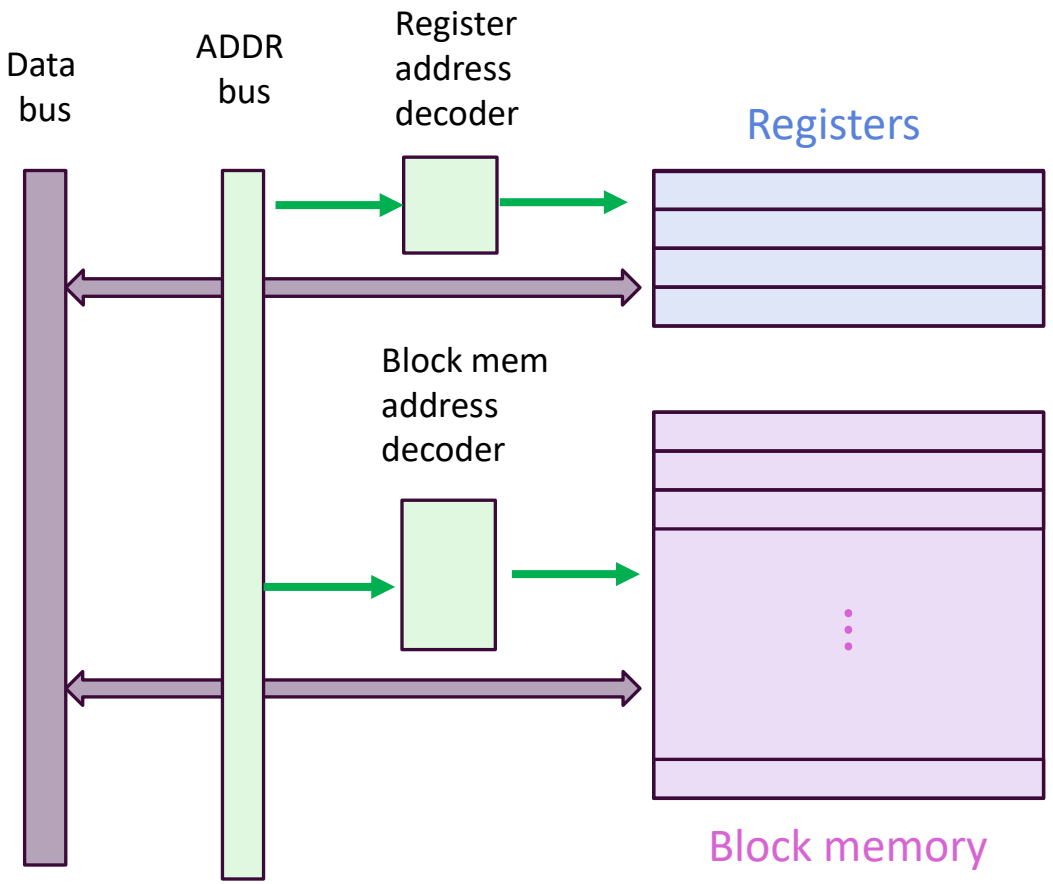
Registers vs. Block Memory

- We have seen two storage types
- Registers
  - Typically implemented as flip-flops
  - Useful for small, individual data pieces
  - Control registers, intermediate calculations, …
- Block memory
  - Typically, DRAM or SRAM
  - Ideal for large arrays
  - Vectors or arrays



Memory Mapped Registers

- Registers can also be memory-mapped
  - Connected to their own address decoder
  - Determines which register to read/write
- Common interface to processor
  - Processor sees registers & block memory together
  - One continuous address space

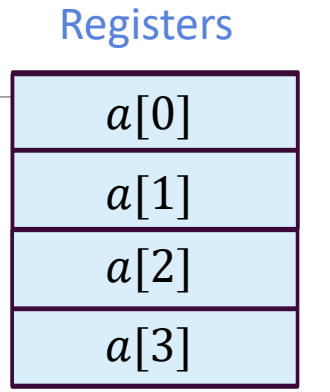


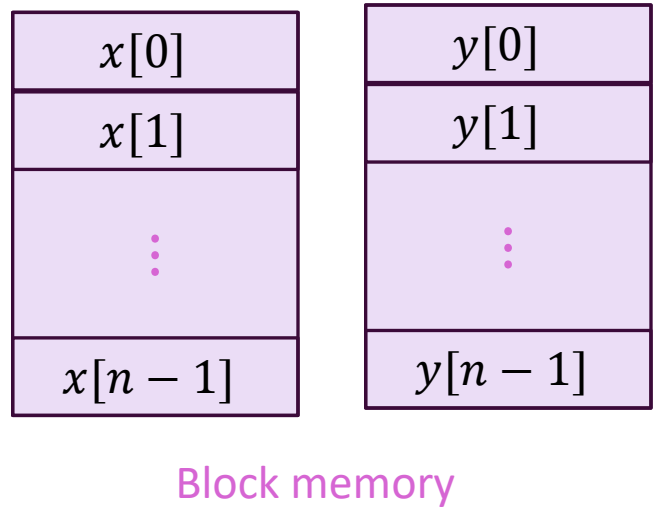
Example

- A HW module computes a cubic function:

$$y_i = a_0 + a_1 x_i + \cdots + a_3 x_i^3$$

- Registers:
  - Store polynomial coefficients $a[0], \ldots, a[3]$
  - Small, individual pieces
  - Want parallel access
  - Could have control registers, e.g. count, data valid, ready, …
- Block memory
  - Store input and output data
  - $x[0], \ldots, x[n-1]$ and $y[0], \ldots, y[n-1]$
  - Ideal for large vectors





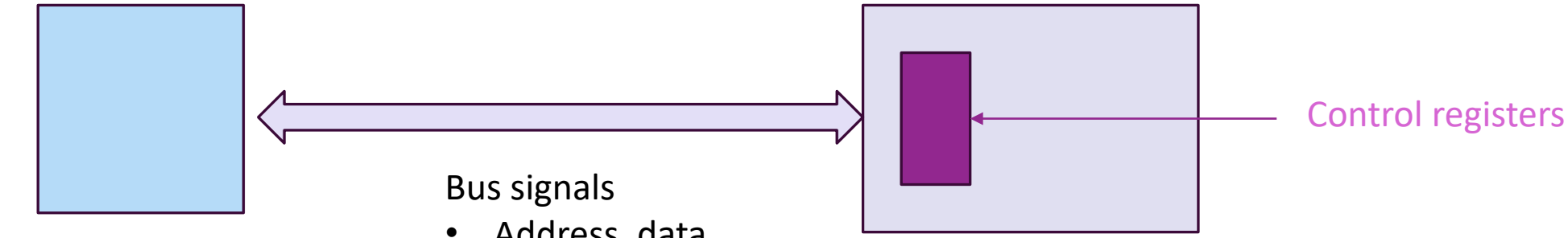
Outline

- Random Access Memory and Memory-Mapped Registers
- Processor Interfaces with AXI-Lite
- Implementing AXI-Lite Interface in Vitis

AXI4-Lite Protocol

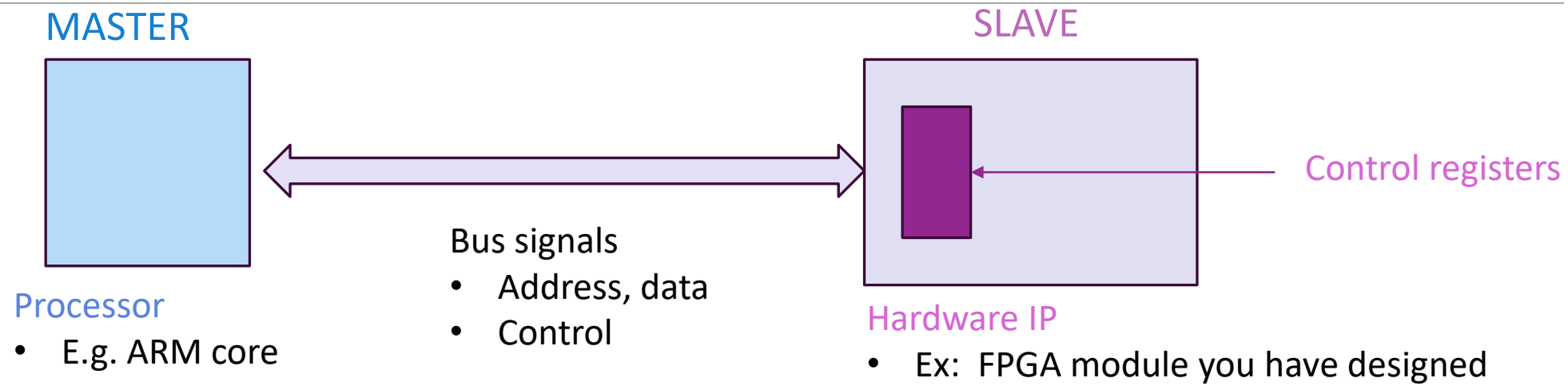
- ARM Extensible Interface family of protocols
  - Others we will use in class are AXI4-Full and AXI4-Stream
- Industry-standard, ubiquitously used
- AXI4-Lite:
  - Simple
  - Ideal for programming registers
  - Modular enables flexibility in delays and scheduling at both ends
  - Typical throughput: 1 read / 1 write every 3-5 clock cycles

AXI4-Lite Use Case: Processor to Hardware IP



- ❑Widely-used use case
- ❑Hardware IP has control registers
- ❑Need read / write access from an embedded processor

Master-Slave and Transactions

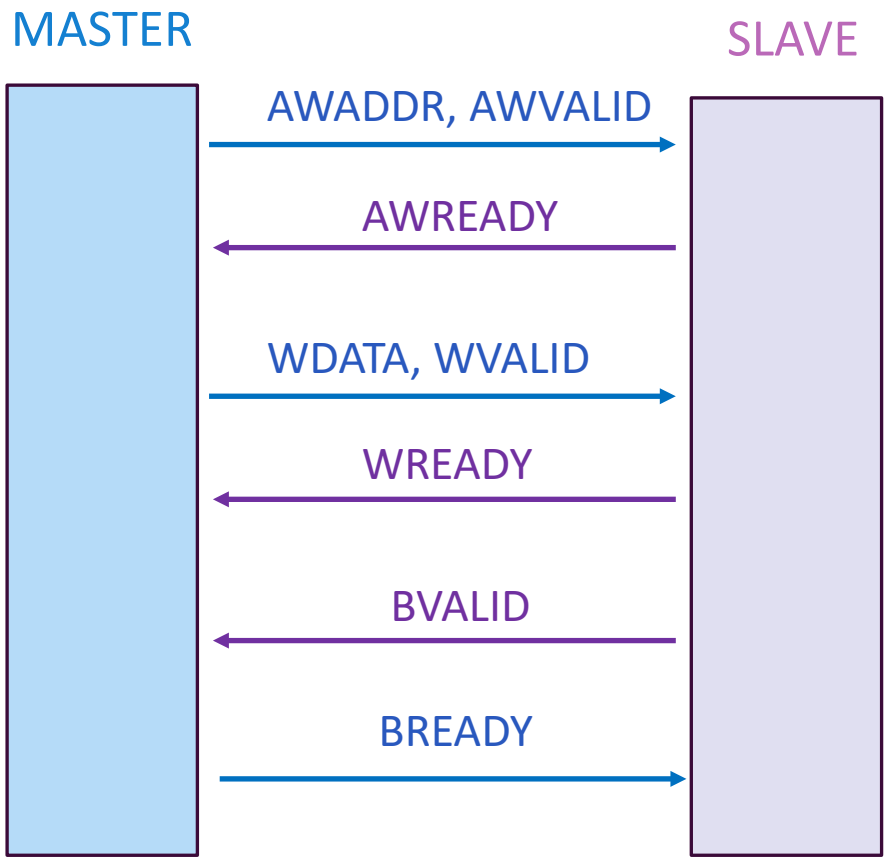


In AXI4-Lite:

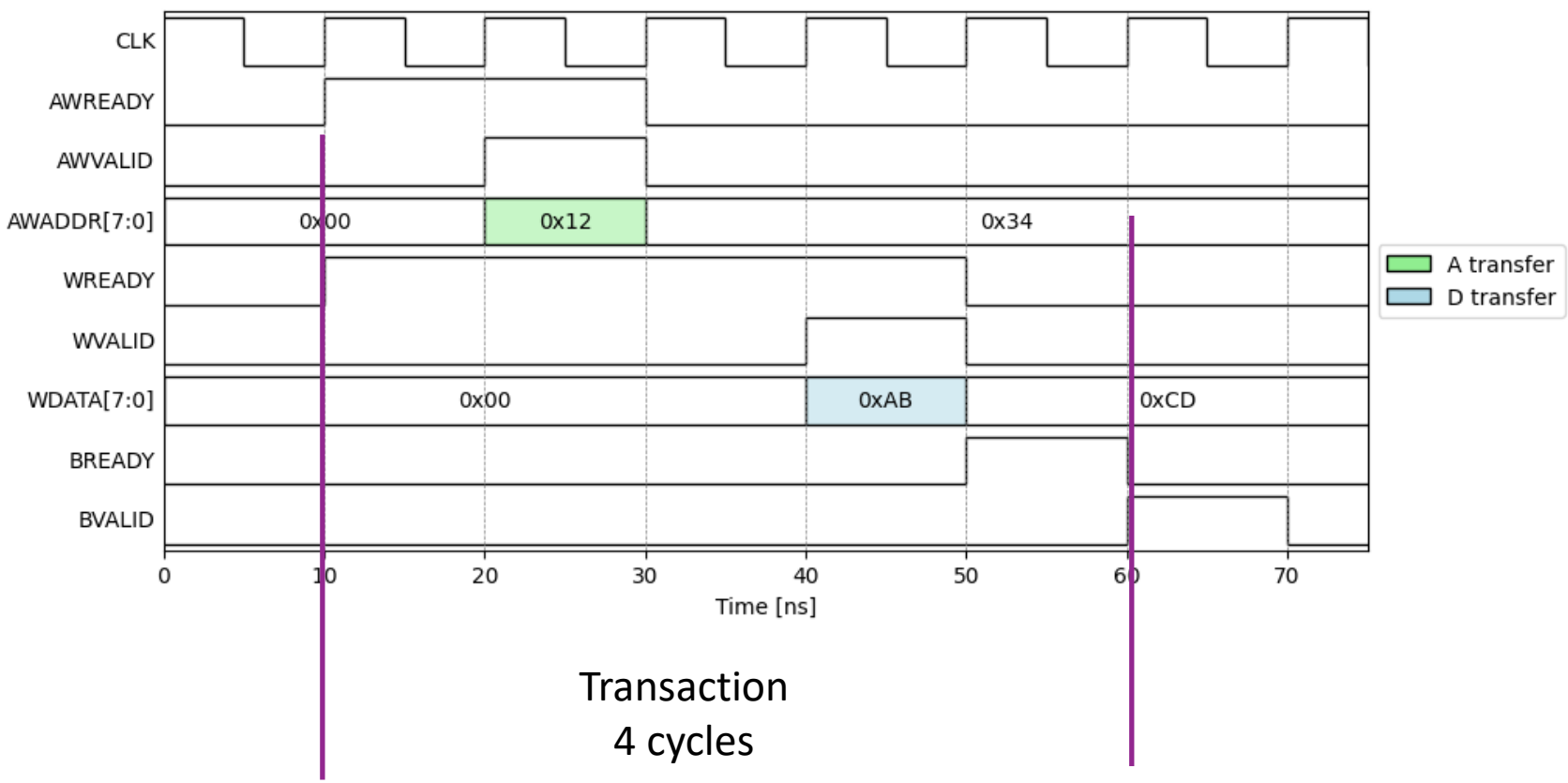
- Each read or write of a single word is a transaction
- Master: The unit that initiates all transaction (typically the processor in this example)
- Slave: The unit that responds to the transaction (typically the hardware IP)

AXI4-Lite Write Protocol

- Suppose master wants to write data D to address A
  - AWADDR=A and AWVALID=1
  - WDATA=D and WVALID=1
  - Both can occur in independent clock cycles
- When slave is ready for address or data it sets:
  - AWREADY=1 and WREADY=1
  - Slave can set these independently and any time it is ready
- Transfer:
  - When AWREADY=AWVALID=1 together address A is transferred
  - When WREADY=WVALID=1 together data D is transferred
- Handshake:
  - After both are transferred, slave set BREADY=1
  - Master sets BVALID=1 to acknowledge
  - Can start next transaction in same clock cycle as BVALID=1



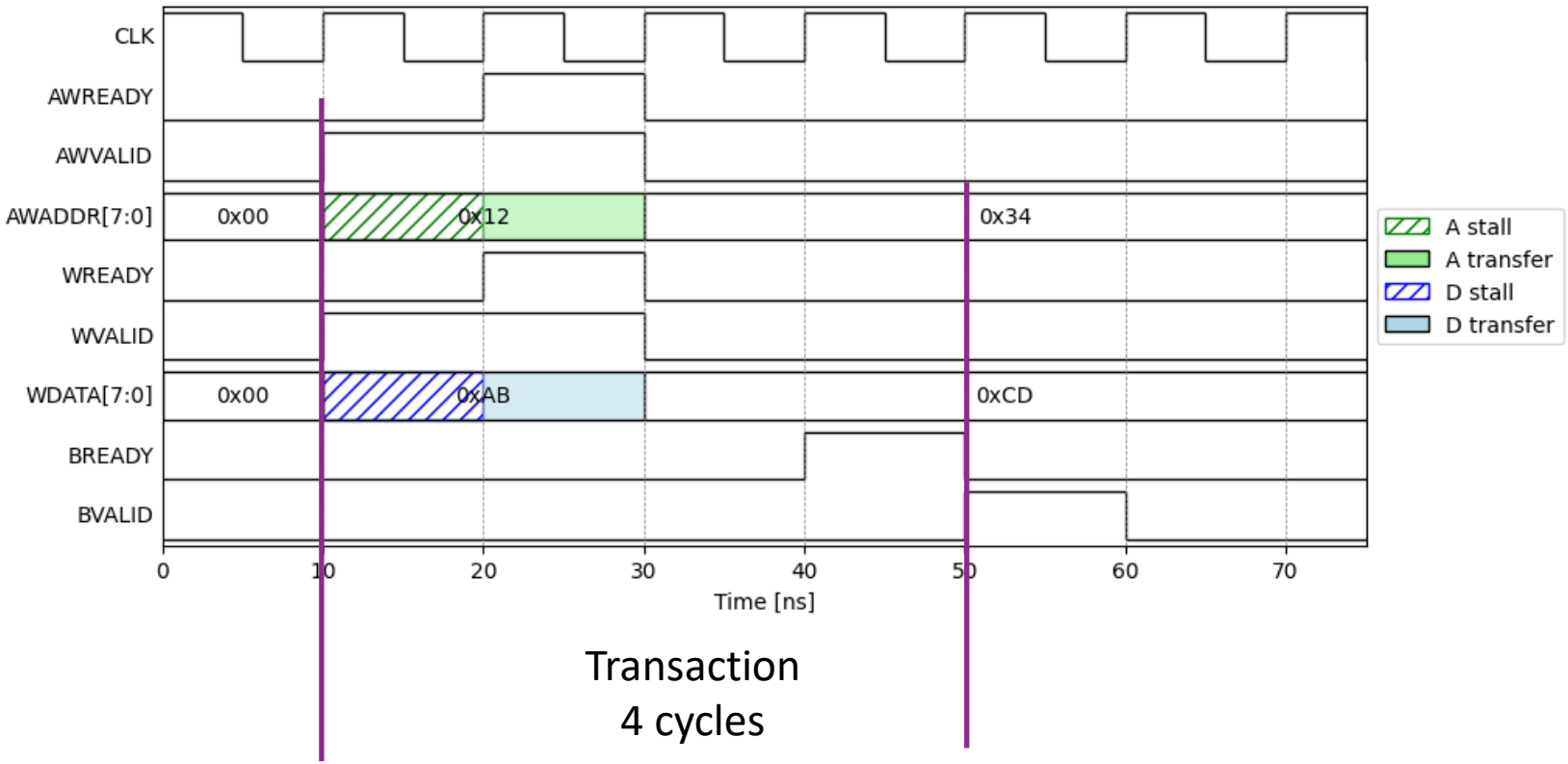
Example AXI4-Lite Write



Example with

- Slave is ready before master
- Address write before data write
- But either order or together is fine

Example AXI4-Lite Write With Stall



Example:

- Master sets valid
- Slave is not ready
- Transfer stalls
- Provides flow control
  - Ensures data waits until RX is ready

Theoretical Optimal Write Throughput

- Cycle 0:
  - Master sets AWVALID=1, AWADDR=A and WDATA=D and WVALID=1
  - Slave is ready: AWREADY=WREADY=1
  - Both address and data are transferred to slave
- Cycle 1:
  - Slave set BVALID=1 signaling transaction complete on slave side
  - Slave registers
- Cycle 2:
  - Master acknowledges with BREADY=1
  - Master starts next transaction in same clock cycle
- Theoretical optimal throughput = 1 word write / 2 clock cycles
- Theoretical minimum latency = 2 clock cycles (master write + slave register)

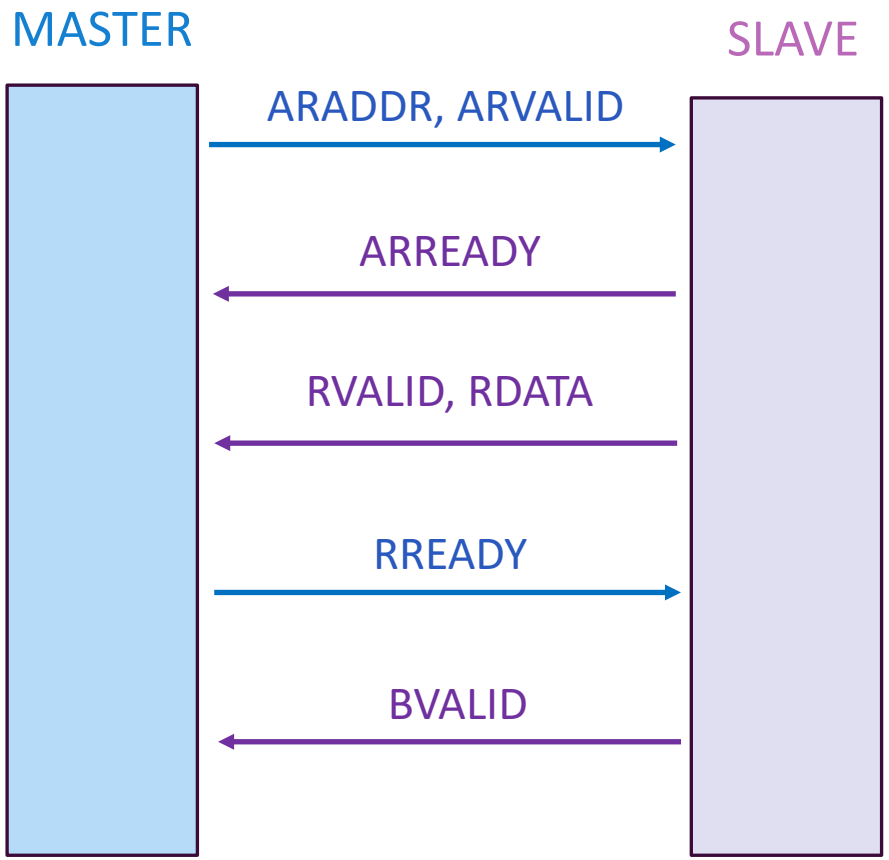
Practical AXI4-Lite Latency

- Suppose:
  - Master sets data and address on cycle 0
  - Slave is ready
- In most practical register systems:
  - Slave takes 1-2 additional clock cycles for BREADY=1
- Master takes at 1-2 additional clock cycles for BVALID=1 and next input

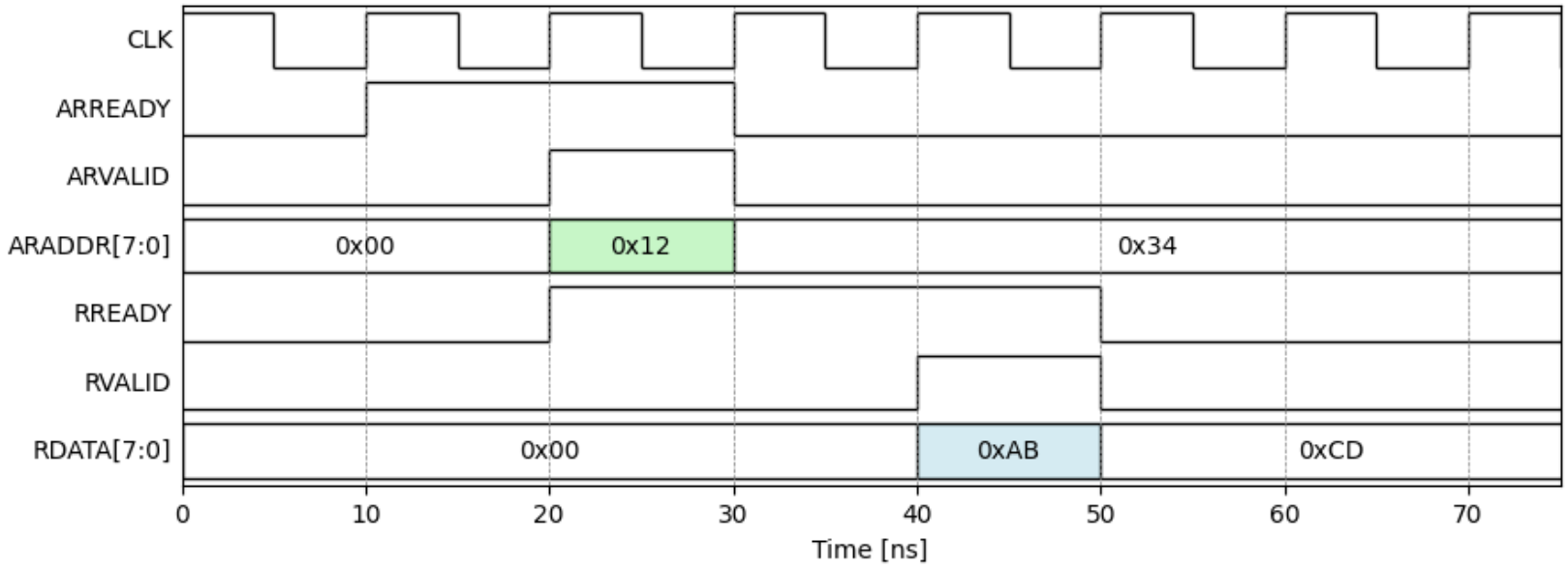
Practical throughput with AXI4-Lite: 1 word write per 3-5 clock cycles

AXI4-Lite Read Protocol

- Suppose master wants to read data from address A
  - ARADDR=A and ARVALID=1
  - Indicates that it wants to read data
- When slave accepts the request:
  - Sets ARREADY=1
- When slave has fetched data:
  - Sets RVALID=1, RDATA=D
  - Can be 0 to many cycles after ARREADY=1
- Handshake:
  - Master asserts RREADY=1 when ready to accept data
  - RREADY could be asserted before RVALID=1
  - Transfer completes when RVALID=RREADY=1



Example Read Timing



In this example:

- Slave is initially ready (ARREADY=1)
- Slave immediately accepts request
- Master is immediately available for
- Data valid with two cycle delay

Theoretical Optimal Read Throughput

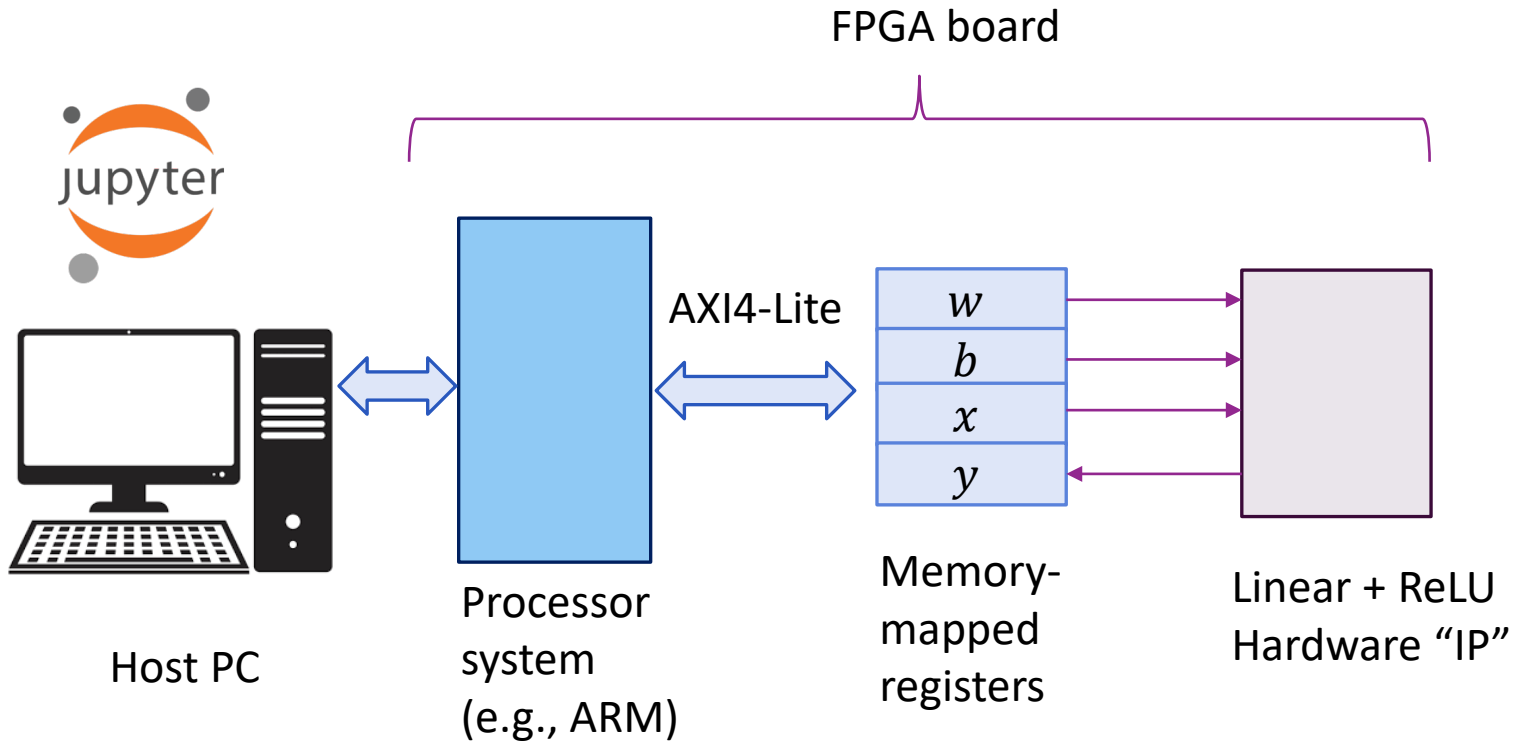
- Cycle 0:
  - Master sets ARADDR=A and ARVALID=1
  - Slave is ready: ARREADY=1
  - Address is transferred to slave
- Cycle 1:
  - Slave set RVALID=1 with data RDATA=D
  - Master is immediately ready, RREADY=1
- Cycle 2:
  - Master moves to next read address
- Theoretical optimal throughput = 1 word read / 2 clock cycles
- Practical throughput = 1 word read per 3-5 clock cycles

Outline

- Random Access Memory and Memory-Mapped Registers
- Processor Interfaces with AXI-Lite
- Implementing AXI-Lite Interface in Vitis HLS

Example: Connecting a Simple IP

- ReLU + Linear $y = \max\{wx + b, 0\}$
  - Identical to Unit 1
- Add an interface to a processor
  - Processor can be an embedded
  - Ex: ARM Core on Pynq board
  - Interfaces to the IP over AXI4-Lite
- Add control from host PC
  - Ideally jupyter notebook



HLS vs. RTL

❑RTL: Register Transfer Language
- Use specialized languages (e.g., SystemVerilog)
- Manually schedule action in each clock cycle
- Interfaces designed from scratch

✓ High control, supports custom interfaces

✗ But time-consuming, difficult to maintain

```
module relu_lin (
  input logic int x,
  input logic int a,
  input logic int b,
  output logic int y
);
  always_comb begin
    logic int mult_out, add_out;
    mult_out = x * a;
    add_out = mult_out + b;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

❑HLS: High-Level Synthesis
- Write in general purpose languages (e.g., C++)
- Clock-level schedule determined automatically by compiler
- API support for common interfaces (e.g., AXI)

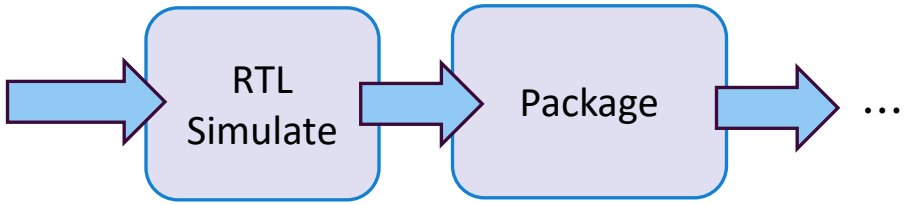
✗ Sacrifice some performance and flexibility

✓ But enables rapid development

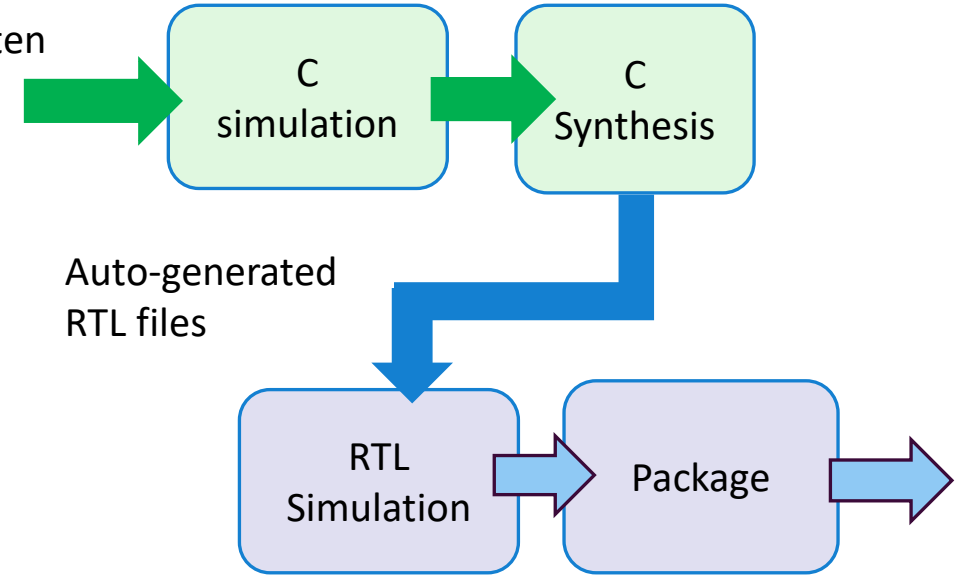
```
void relu_lin(int x, int a, int b, int &y) {
  int mult_out = x * a;
  int add_out = mult_out + b;
  if (add_out > 0) {
    y = add_out;
  } else {
    y = 0;
  }
}
```

HLS vs. RTL FPGA Design Flow

Human-written
RTL files
(e.g. SV)



Human-written
C files

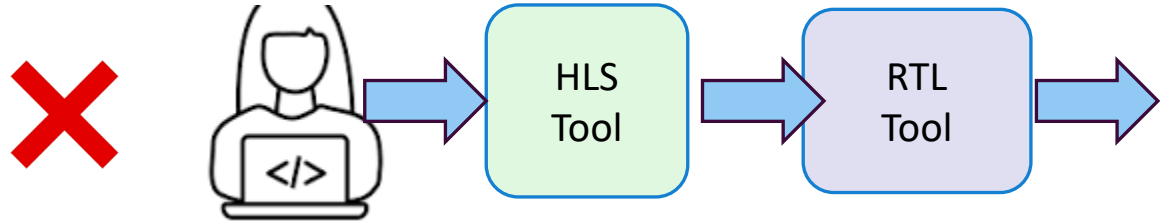


- Traditional RTL Design Flow
  - Designer directly writes RTL
  - Tools simulate and synthesize
- HLS Flow
  - Designer write high-level language (e.g., C)
  - Standard software compile-synthesis
  - RTL files auto-generated

Bad vs. Good HLS Code Write

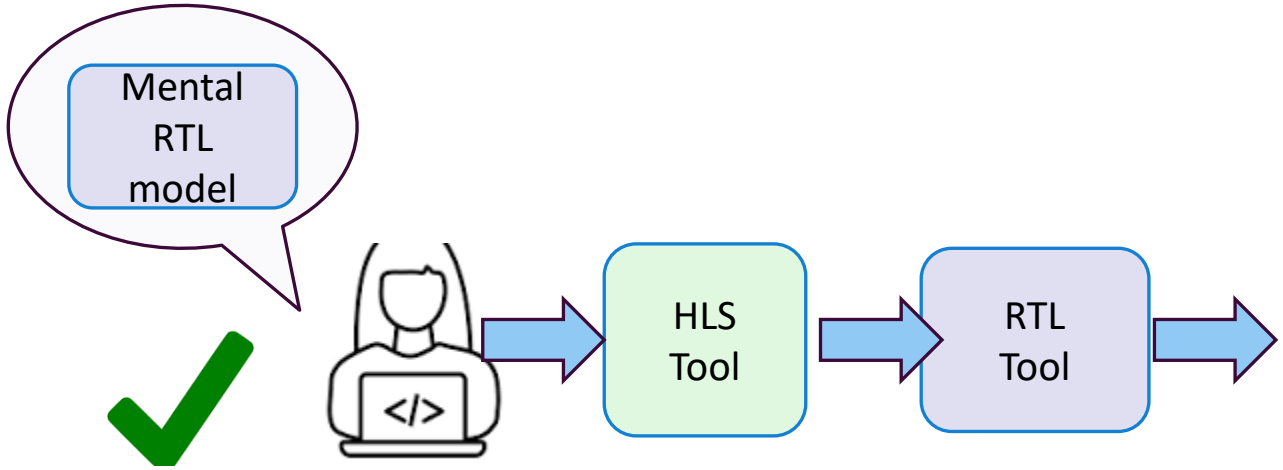
Bad: Treat HLS tool as a black box

- ❑Results in RTL code with poor performance
- ❑No way to evaluate if resulting design is good



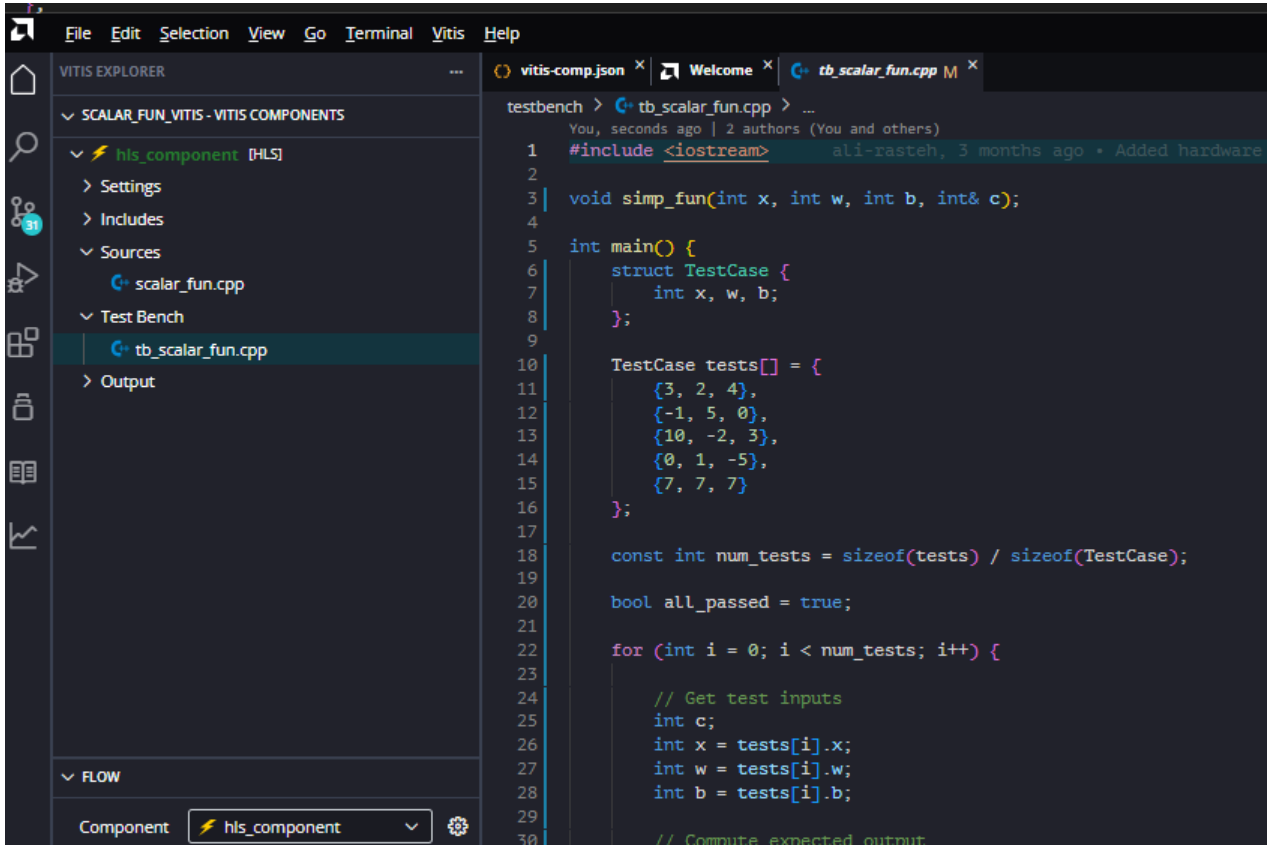
Good: Have a mental model

- ❑Think what the RTL code should look like
- ❑What should happen on each clock cycle
- ❑What resources should be used



Vitis HLS

- Developed by Xilinx (now AMD)
  - Evolved from Vivado HLS
- Supports C / C++
  - Simple, common development
- Some python support for testbenches
- Ships with Vitis IDE
- Native support for AXI protocols
  - No need to build handshake from scratch



FPGA Board Components

- We target a design on a real FPGA board
- Boards have three components
- Processing system (PS)
  - A lightweight processor
  - Often an ARM core
  - E.g., ARM A9
- Programmable logic (PL)
  - The programmable fabric
  - This is where your IP will reside
  - Also contains the registers and internal memory
- An AXI4 interface

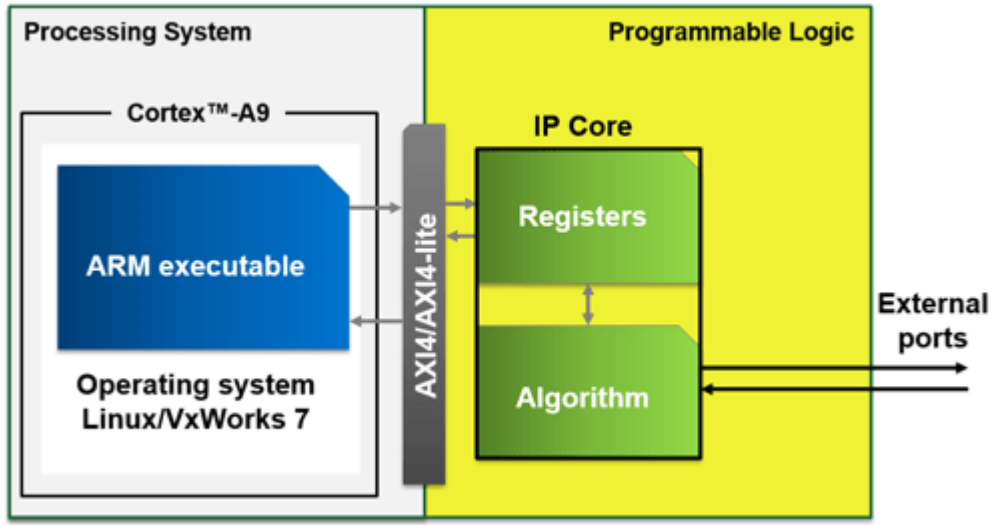
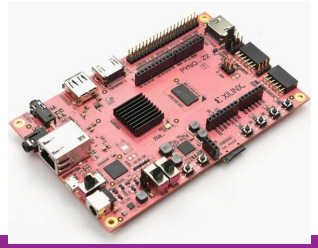
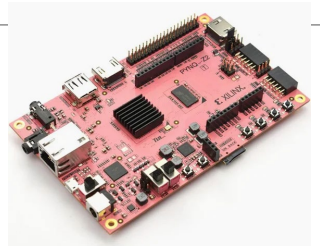


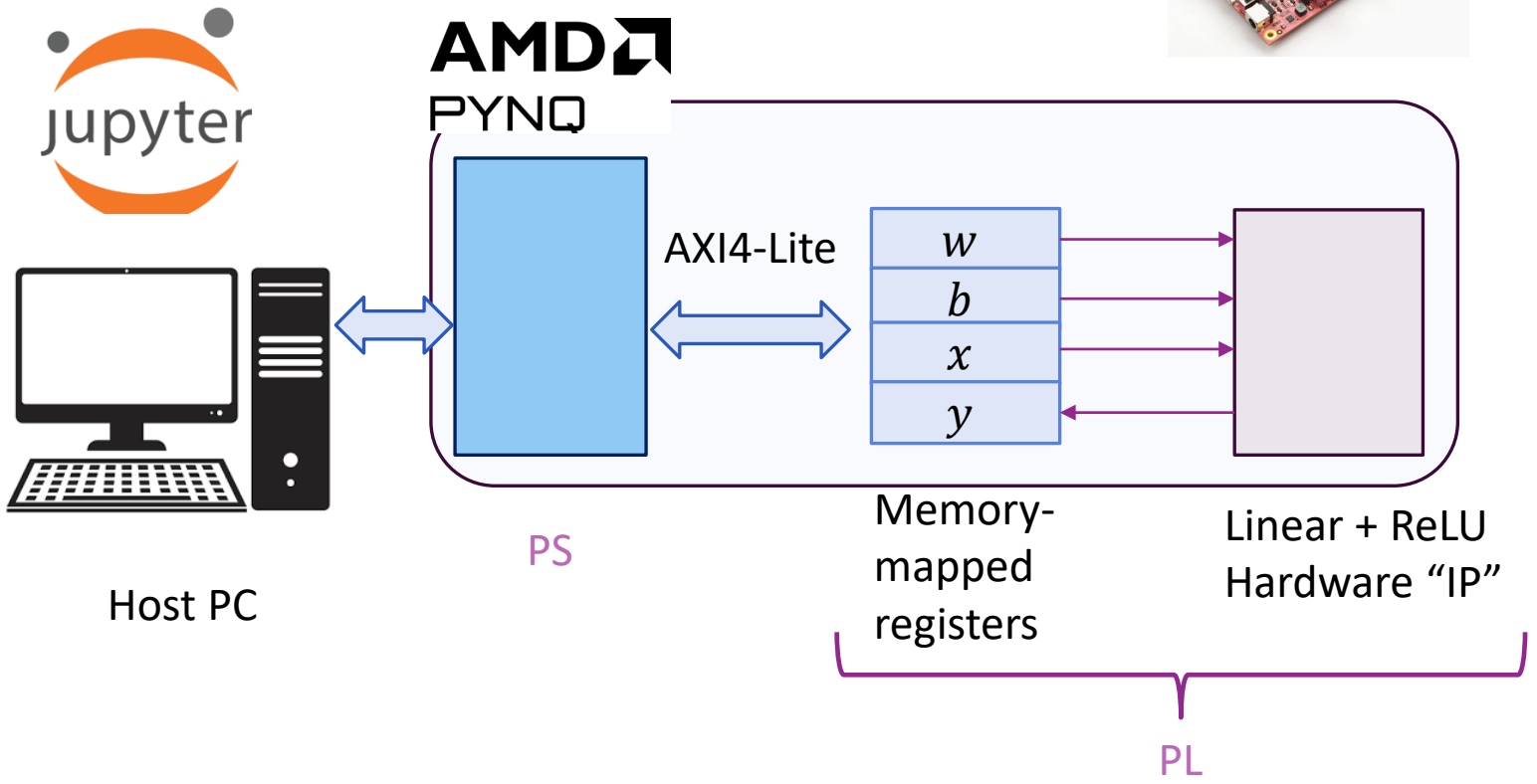
Image from MathWorks page on embedded design



Target Architecture

Pynq Z2 board





We will rapidly build

- ❑ Complete AXI4-Lite I/F
  - Interface done automatically
  - No need to code any handshaking
- ❑ Control through host
  - Simple python jupyter notebook
- ❑ Deploy to a real FPGA!

IP Source Code

```
void simp_fun(int x, int w, int b, int& y) {

    #pragma HLS INTERFACE s_axilite port=x      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=w      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=b      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=y      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL

    int act_in = w * x + b;
    if (act_in > 0)
        y = act_in;
    else
        y = 0;
    }
```

IP Source Code is simple

- Module declaration
- Port descriptions
- Main functional description

Module Declaration

```
void simp_fun(int x, int w, int b, int& y) {

    #pragma HLS INTERFACE s_axilite port=x      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=w      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=b      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=y      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL

    int act_in = w * x + b;
    if (act_in > 0)
        y = act_in;
    else
        y = 0;
    }
```

Module declaration

- Defines inputs and outputs
- Standard C

Module Declaration

```
void simp_fun(int x, int w, int b, int& y) {

    #pragma HLS INTERFACE s_axilite port=x      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=w      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=b      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=y      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL

    int act_in = w * x + b;
    if (act_in > 0)
        y = act_in;
    else
        y = 0;
    }
```

Port definitions

- Use pragma statements
- Do not change function
- But tell synthesizer how to map to hardware
- Indicates ports are axislite (AXI4-Lite slave)
- Share a common bundle
  (all on the same AXI4-Lite interface)
- Additional port=return
  (Provides start, stop, see later)

Module Declaration

```
void simp_fun(int x, int w, int b, int& y) {

    #pragma HLS INTERFACE s_axilite port=x      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=w      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=b      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=y      bundle=CTRL
    #pragma HLS INTERFACE s_axilite port=return bundle=CTRL

    int act_in = w * x + b;
    if (act_in > 0)
        y = act_in;
    else
        y = 0;
    }
```

Main function definition

- Plain C
- Do not specify the implementation
  Just the desired behavior
- Synthesizer may map to multiple clock cycles
- You do not have to do scheduling manually

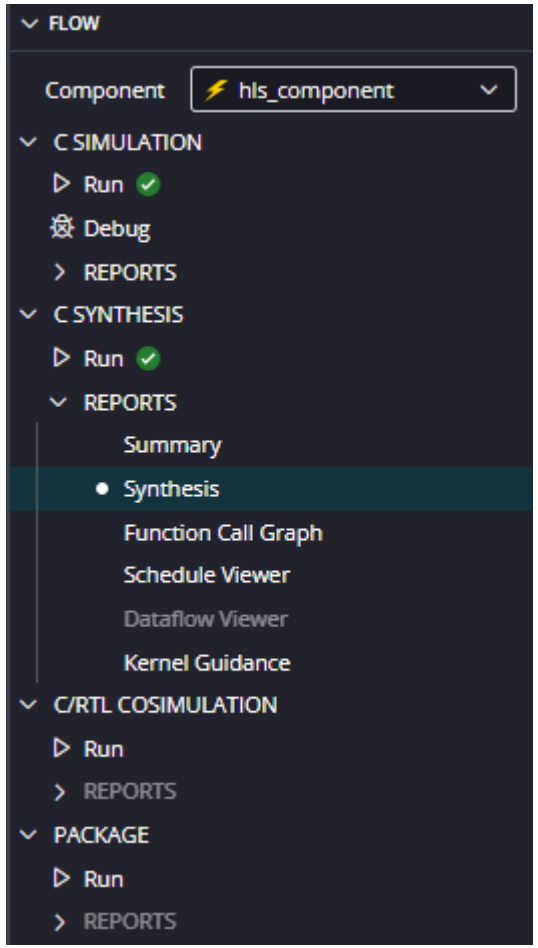
Vitis HLS Testbench

- Testbench
  - Program simulating another unit interacting with IP
  - IP is the device-under-test (DUT)
- In HLS:
  - Testbench another C program
  - Calls the DUT module like a C function
  - Sequential behavior
  - Testbench code can use any C / C++ functions
  - Does not need to be “synthesizable”

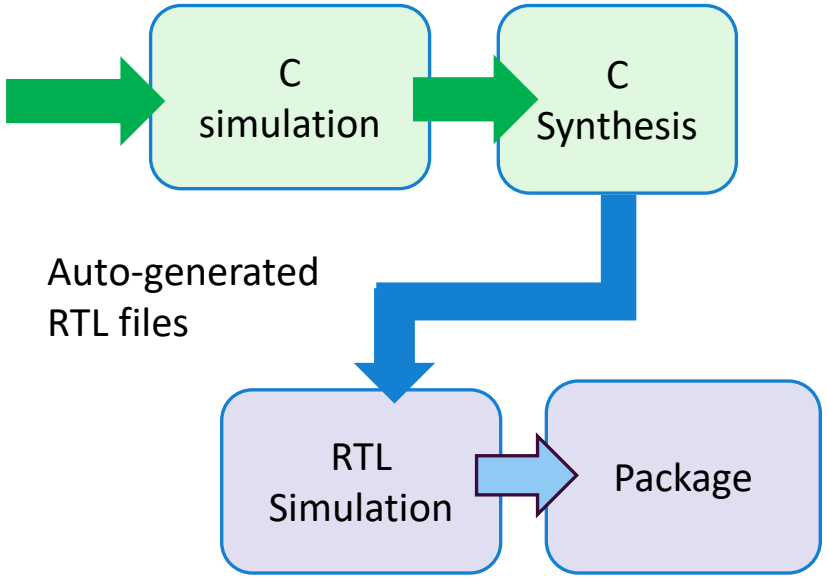
```
int main() {
    struct TestCase {
        int x, w, b;
    };

    TestCase tests[] = {
        {3, 2, 4},
        {-1, 5, 0},
        {10, -2, 3},
        {0, 1, -5},
        {7, 7, 7}
    };
```

```
for (int i = 0; i < num_tests; i++) {

    // Get test inputs
    int c;
    int x = tests[i].x;
    int w = tests[i].w;
    int b = tests[i].b;

    // Compute expected output
    int c_exp = w * x + b;
    if (c_exp < 0) c_exp = 0;

    simp_fun(x, w, b, c);


    std::cout << "Test " << i << " got " << c << ", expected " << c_exp;
    if (c != c_exp) {
        std::cout << " FAIL" << std::endl;
        all_passed = false;
    }
    else {
        std::cout <<  " PASS" << std::endl;
    }

if (all_passed)
    std::cout << "All tests passed!" << std::endl;
else
    std::cout << "Some tests failed." << std::endl;
}
```

Vitis HLS Flow Described in Flow Panel



- Tool follows expected steps for HLS flow



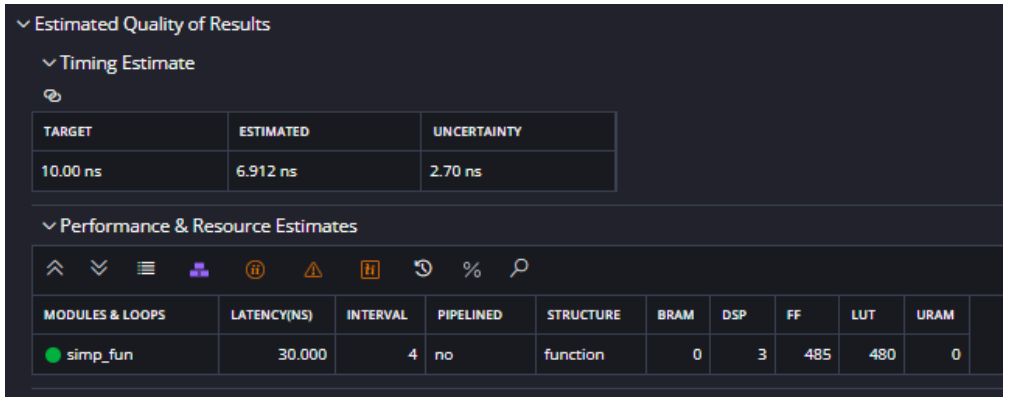
Synthesis Output

- Resulting Verilog files are massive!
- Main function in this case:
  - simp_fun.v
  - > 200 lines of Verilog
  - Hard to read code
- You saved a lot of time coding

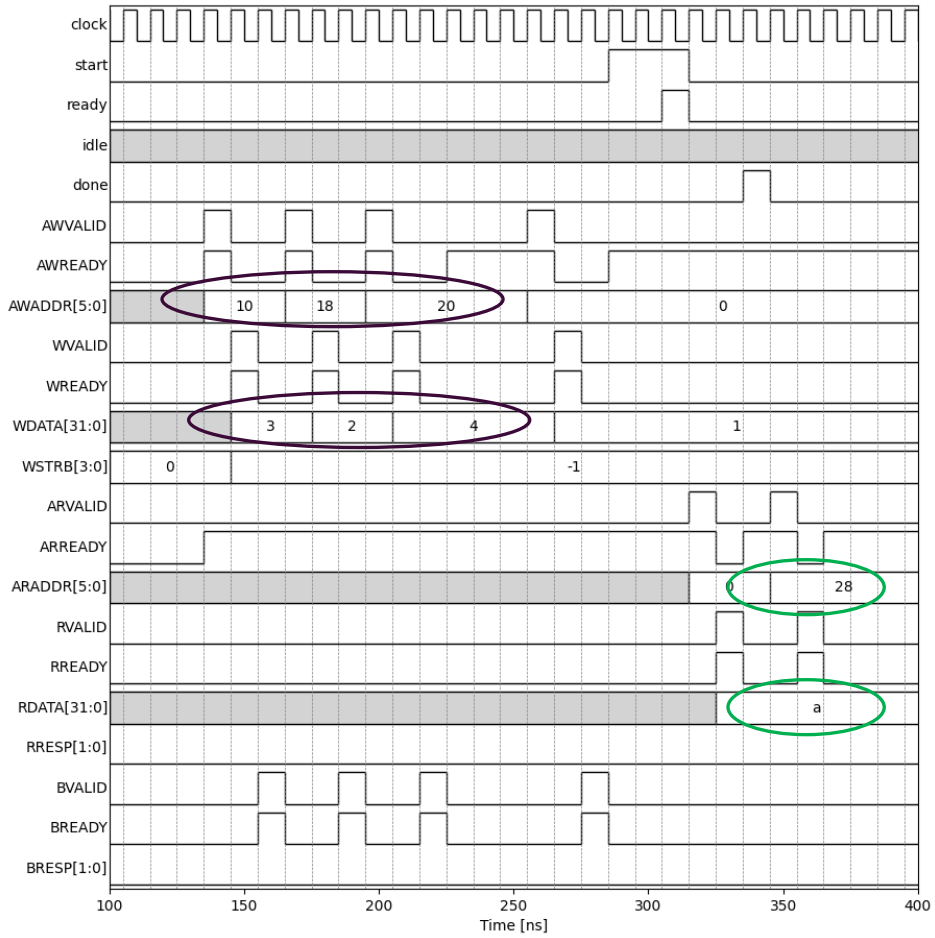
```
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="simp_fun_simp_fun,hls_i

module simp_fun (
        ap_clk,
        ap_rst_n,
        s_axi_CTRL_AWVALID,
        s_axi_CTRL_AWREADY,
        s_axi_CTRL_AWADDR,
        s_axi_CTRL_WVALID,
        s_axi_CTRL_WREADY,
        s_axi_CTRL_WDATA,
        s_axi_CTRL_WSTRB,
        s_axi_CTRL_ARVALID,
        s_axi_CTRL_ARREADY,
        s_axi_CTRL_ARADDR,
        s_axi_CTRL_RVALID,
        s_axi_CTRL_RREADY,
        s_axi_CTRL_RDATA,
        s_axi_CTRL_RRESP,
        s_axi_CTRL_BVALID,
        s_axi_CTRL_BREADY,
        s_axi_CTRL_BRESP,
        interrupt
);

parameter    ap_ST_fsm_state1 = 4'd1;
parameter    ap_ST_fsm_state2 = 4'd2;
parameter    ap_ST_fsm_state3 = 4'd4;
parameter    ap_ST_fsm_state4 = 4'd8;
parameter    C_S_AXI_CTRL_DATA_WIDTH = 32;
parameter    C_S_AXI_CTRL_ADDR_WIDTH = 6;
parameter    C_S_AXI_DATA_WIDTH = 32;

parameter C_S_AXI_CTRL_WSTRB_WIDTH = (32 / 8);
parameter C_S_AXI_WSTRB_WIDTH = (32 / 8);
```

Synthesis Report

- Generate a Report in Vitis Tool
  - Later, we will write python to parse these
- Timing estimate
  - Timing of critical path
  - Can verify that lower than clock period
- Estimated latency
  - In this case = 4 clock cylces
- Estimate resources
  - Amount of each FPGA resources used
  - BRAM, DSP, …
  - We will discuss these in detail later

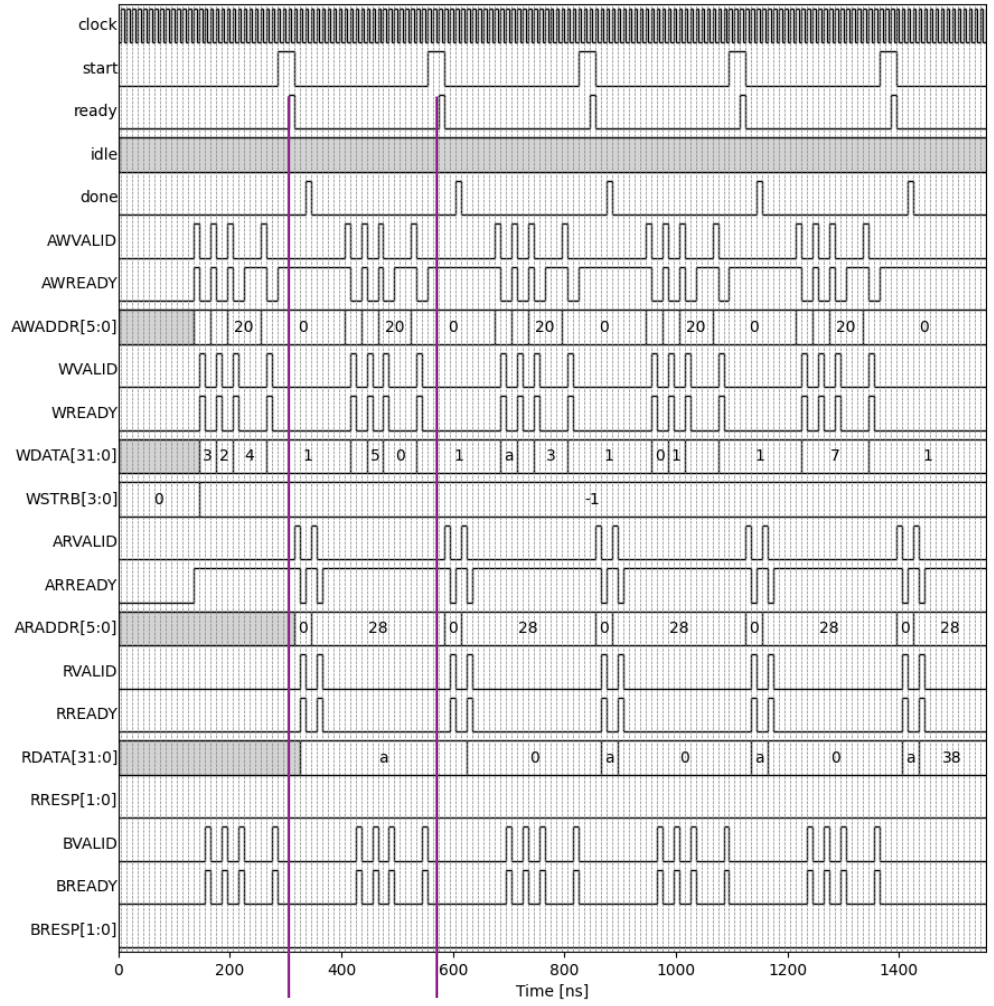


RTL Simulation



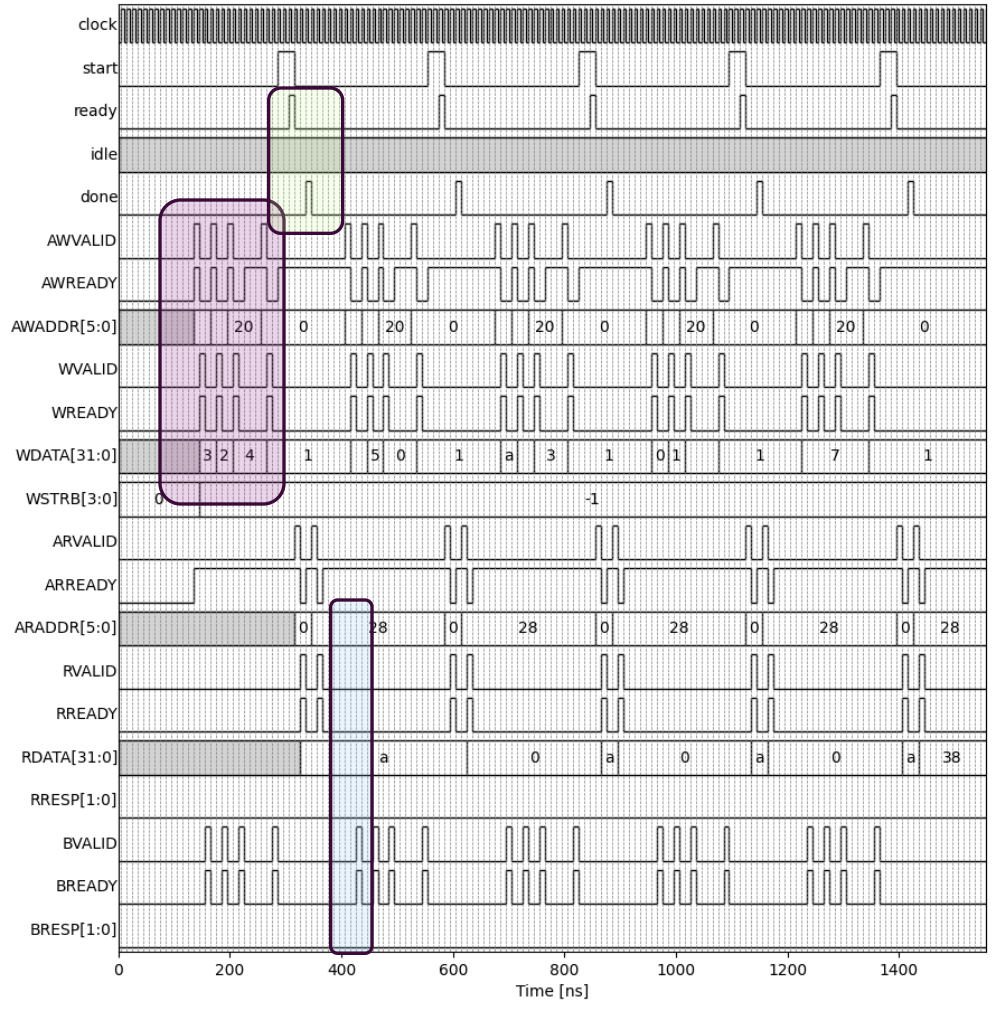
- Run RTL sim and extract VCD diagram
- Left plot:
  - One transaction with IP
  - Can visualize AXI4-Lite interaction
- Write:
  - Write three values: w, x, b
  - 3 clock cycles each
- Read:
  - Reads output
  - Approximately 20 cycles after first write

RTL Simulation over Multiple Interactions



- Approximately 20 cycles for each input
- We will be able to do this much faster
  - Will use an AXI-Streaming interface
  - Next lecture
- AXI4-Lite is design for slowly varying items

Execution Model



- Write phase
  - PS writes input control registers
  - PS "start" on the control register
  - Transaction starts when "ready"
  - IP asserts "done"
  - PS reads outputs

Deploying on an FPGA Board

- Will show this in class