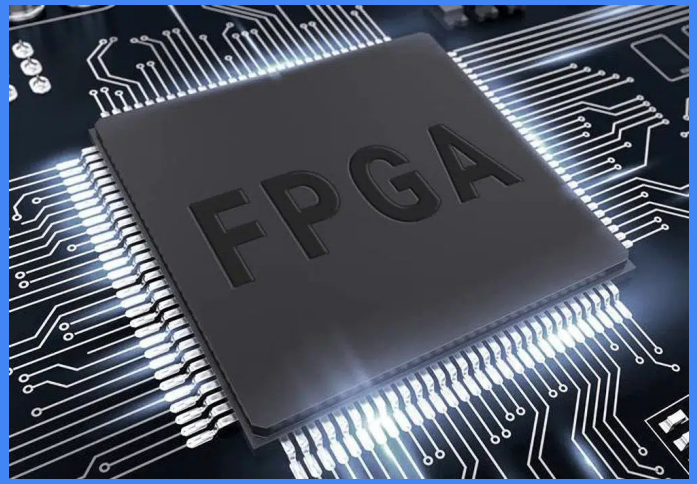



# Some more HLS stuff

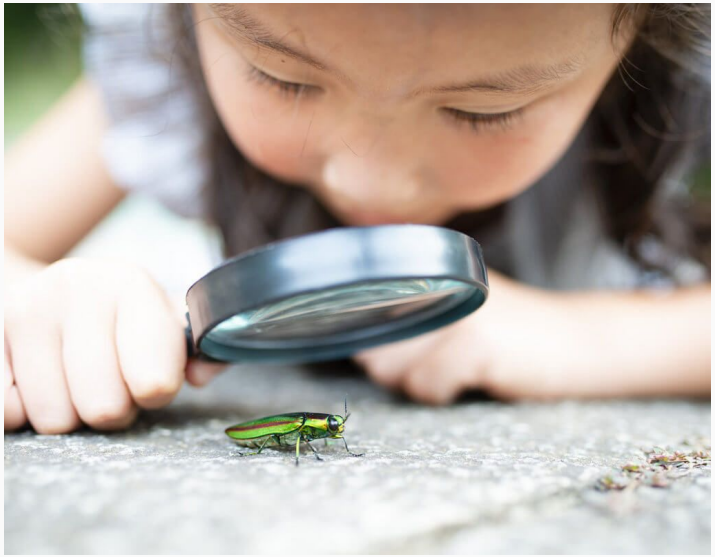
By Jahred, with lots of material from
Jovan Mitrevski (FNAL) and from the Vitis HLS User Guide and other places

# A reminder of what HLS is doing

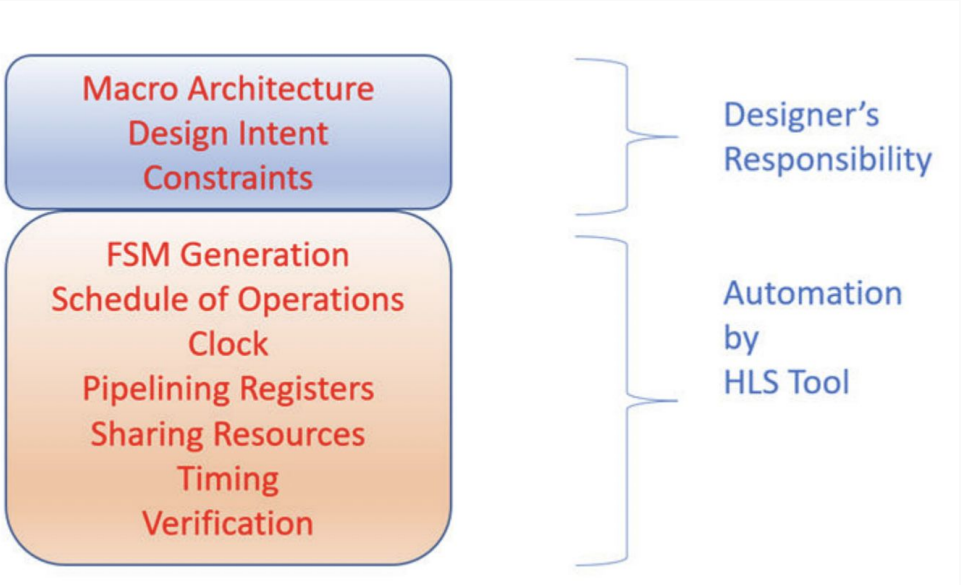
- We want to **program our FPGA to do our bidding** (read: self-drive our cars and/or trigger on new physics events!), and to do so we need to create RTL code
- **HLS offloads the creation of RTL to the compiler**, leaving us to write (mostly) familiar C-style code!
- Can ask the compiler to hit **target timing constraints**, and using ***pragmas*** and other tools we can trade off resource usage vs latency and speed

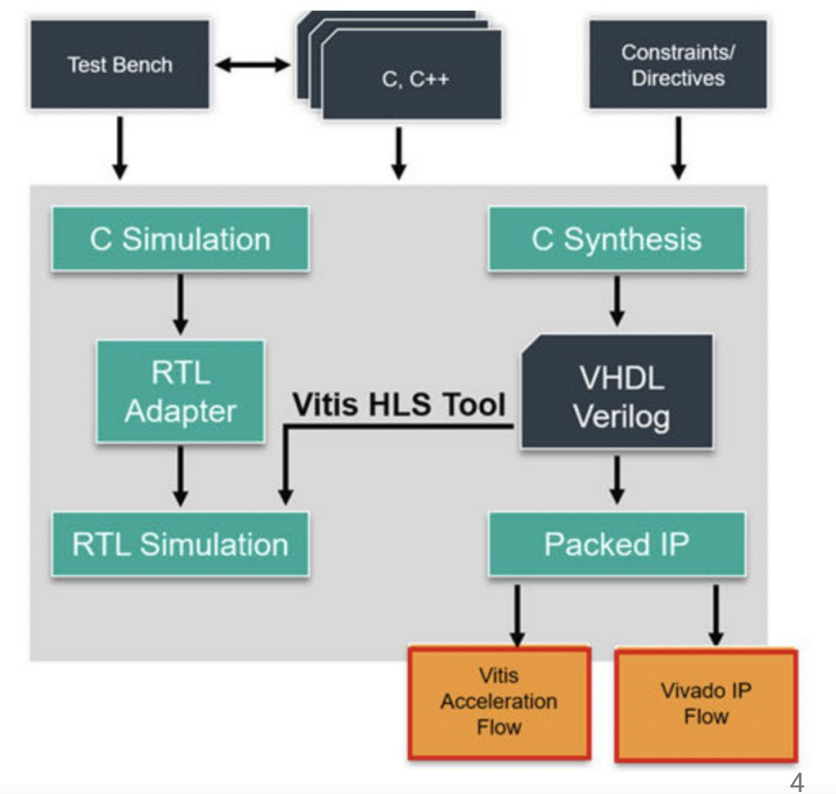
# A huge advantage to HLS

- We always want to **validate our HLS code**, in particular for complicated algorithms
- Having a **test bench for validation** is useful, but we may want to check the bigger picture
  - In other words, assuming no bugs, **does this algorithm do what we think**? Checking that in RTL is not simple
  - Checking your HLS code is much easier - it is already close to standard C code!



# Design and development from Vitis HLS manual



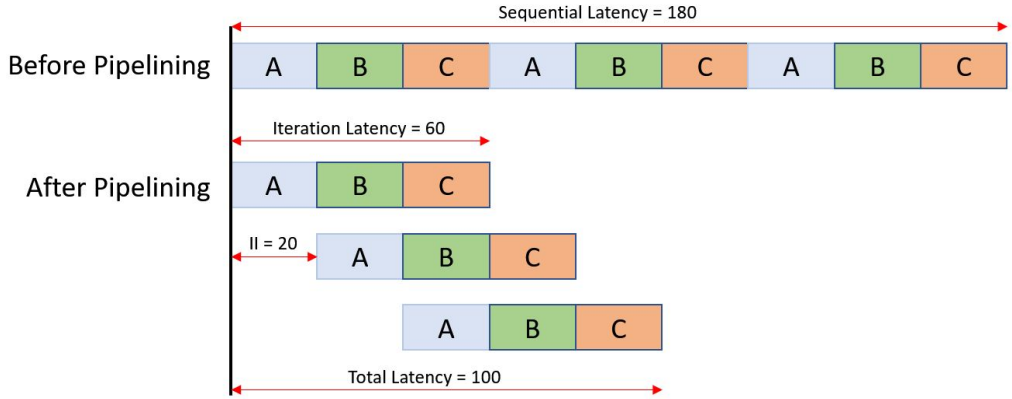
# More re: HLS

- The HLS **design ultimately results in a circuit**
  - There is no OS or runtime system for the program to use
- Many features we know and love about C and C++ are not supported:
  - Dynamic memory, virtual functions, OS calls, **std::vector are no-nos**
- Some useful things are supported
  - Templates, constexpr, C arrays and std::array, compile-time recursion
  - Pointers are supported but not really recommended
- Rule of thumb: **Generally need to know sizes of objects in advance**
  - Why? Aside from the small available memories on chip, we are not allocating RAM. We are literally storing our variables inside of CLBs and physical logic gates! Cannot just allocate more RAM or use virtual memory
- **Pragmas** guide synthesis of the RTL and change the actual circuit logic

# Pipelining

- As discussed earlier, we will want to **trade off resources for performance** and **parallelize** as much as possible
- **Task A: 20 compute cycles**, **Task B: 10 compute cycles**, **Task C: 30 compute cycles**.

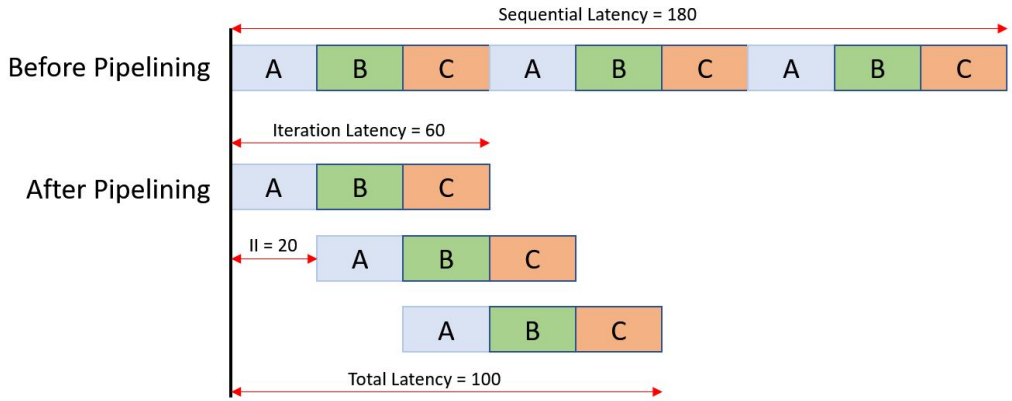
If we do nothing, it take 60 cycles for the first data object to finish, and then we get another one out every 30 cycles



# Pipelining

- If we can **pipeline and run A, B and C in parallel**, we do NOT decrease the **latency**, the time for a single object to be produced = 60 cycles
- But with pipelining, we take only 20 cycles for the second and third objects to be produced after the first time (this is the **Initiation Interval = II**, which is confusing because I looks like an L or the number 1!)

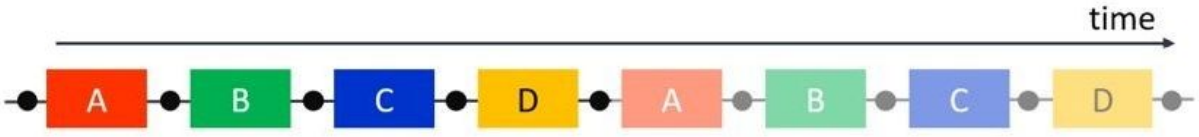
With pipelining, we output 3 objects in 100 cycles, not 180 cycles!



# Serial evaluation

- Objects are evaluated each time through the for loop
  - **Note that functions B and C are independent of one another. No need to execute C after B!**

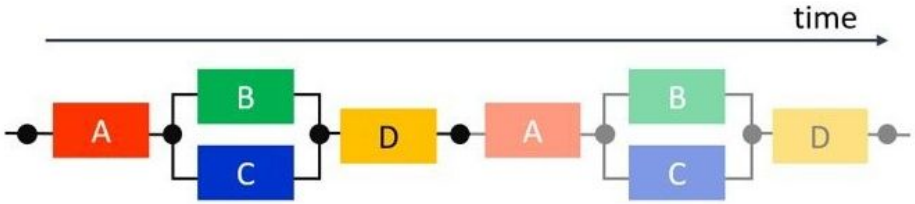
```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
   data_t c1[N], c2[N], c3[N], c4[N];
   #pragma HLS dataflow
   A(vecIn, c1, c2);
   B(c1, c3);
   C(c2, c4);
   D(c3, c4, vecOut);
}
```



# Better implementation


Prefer for evaluation of functions in our for loop to look more like this - **better performance**!

```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
   data_t c1[N], c2[N], c3[N], c4[N];
   #pragma HLS dataflow
   A(vecIn, c1, c2);
   B(c1, c3);
   C(c2, c4);
   D(c3, c4, vecOut);
}
```

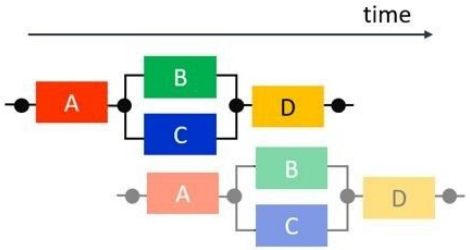


# Even better implementation


This is even better data flow and performance (**smaller overall latency for multiple inputs**)

We **need the compiler to be careful** so that B and C don't read data from A that is being overwritten (ideally Vitis understands and deals with this, not us!)

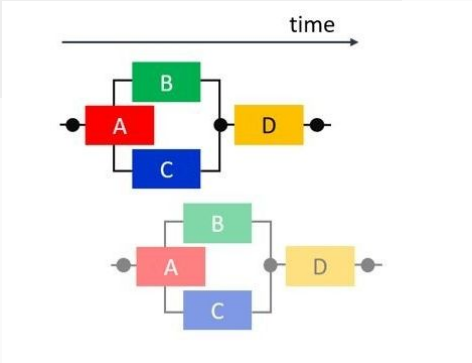
```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
   data_t c1[N], c2[N], c3[N], c4[N];
   #pragma HLS dataflow
   A(vecIn, c1, c2);
   B(c1, c3);
   C(c2, c4);
   D(c3, c4, vecOut);
}
```



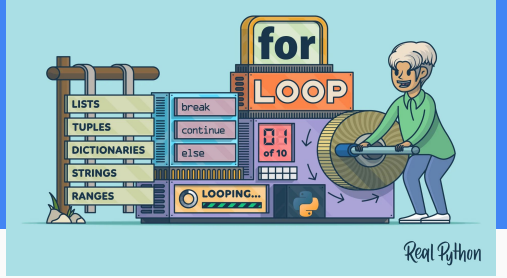
# Best implementation


Need communication between functional blocks to take place via FIFOs so data is not overwritten

```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
    data_t c1[N], c2[N], c3[N], c4[N];
    #pragma HLS dataflow
    A(vecIn, c1, c2);
    B(c1, c3);
    C(c2, c4);
    D(c3, c4, vecOut);
}
```



# For loops

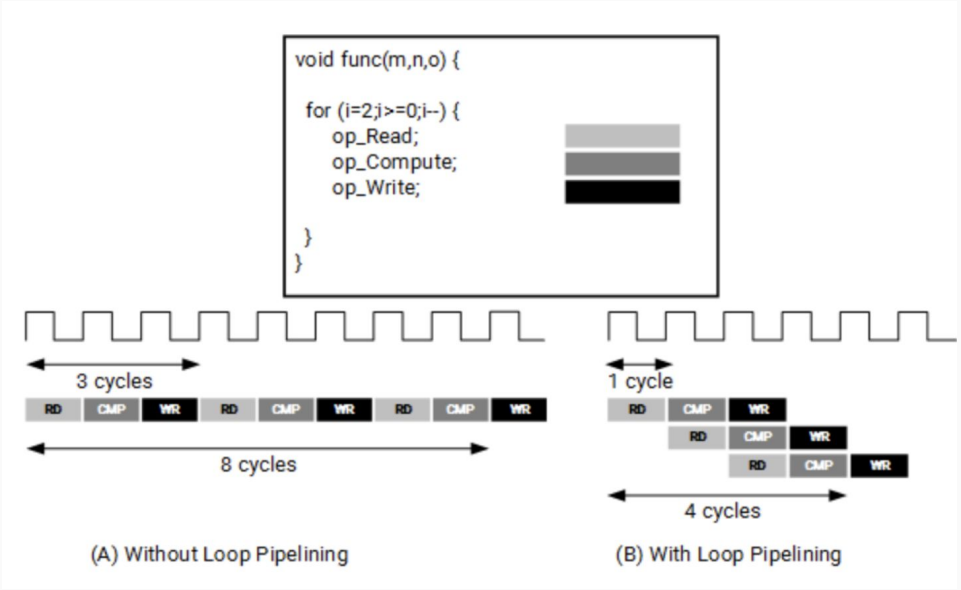



- **For loops are by default a serial construct** - objects are evaluated each time through the for loop
  - See: Your PyROOT for loops grinding your analysis code to a halt

```
vadd: for(int i = 0; i < len; i++) {
    c[i] = a[i] + b[i];
}
```

**If each loop takes t compute cycles then this entire loop takes t*len cycles to complete. Not ideal!**

# Loop pipelining




**Try and use more resources to minimize the initiation interval (II). An II = 1 is ideal, but sometimes not possible. You can specify the II goal and Vitis will try and reach that and will issue a warning if it's not possible. Depending on your settings Vitis will automatically try to do this for you unless you ask it not to**

# Un-pipelineable code

```
Minim_Loop: while (a != b) {
if (a > b)a -= b;
else b -= a;
}
```

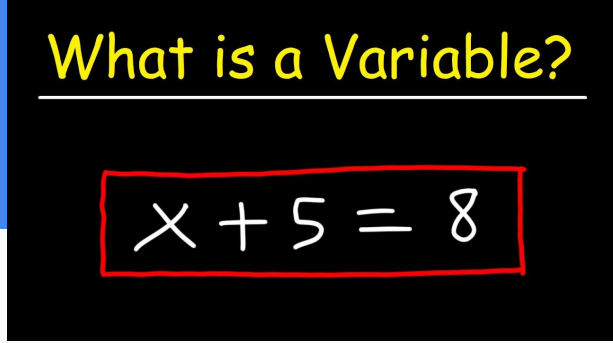
**Cannot know what a and b are until the loop finishes, so you cannot pipeline this!**

# Nested for loops

```
Perfect_nested_loop_1: for (int i = 0; i < N; ++i) {
    Perfect_nested_loop_2: for (int j = 0; j < M; ++j) {
        // Perfect Nested Loop Code goes here and no where else
    }
}

Imperfect_nested_loop_1: for (int i = 0; i < N; ++i) {
    // Imperfect Nested Loop Code contains code here
    Imperfect_nested_loop_2: for (int j = 0; j < M; ++j) {
        // Imperfect Nested Loop Code goes here
    }
    // Imperfect Nested Loop Code may contain code here as well
}
```

**Ideally try and put all your code in nested loops inside the inner-most loop!**

# Variable loop bounds




```
#include "ap_int.h"
#define N 32

typedef ap_int<8> din_t;
typedef ap_int<13> dout_t;
typedef ap_uint<5> dsel_t;

dout_t code028(din_t A[N], dsel_t width) {

 dout_t out_accum=0;
 dsel_t x;

 LOOP_X:for (x=0;x<width; x++) {
 out_accum += A[x];
 }

 return out_accum;
}
```

**Here, Vitis doesn't know how big the for loop will be when it compiles and synthesizes your code. Cannot optimize!**

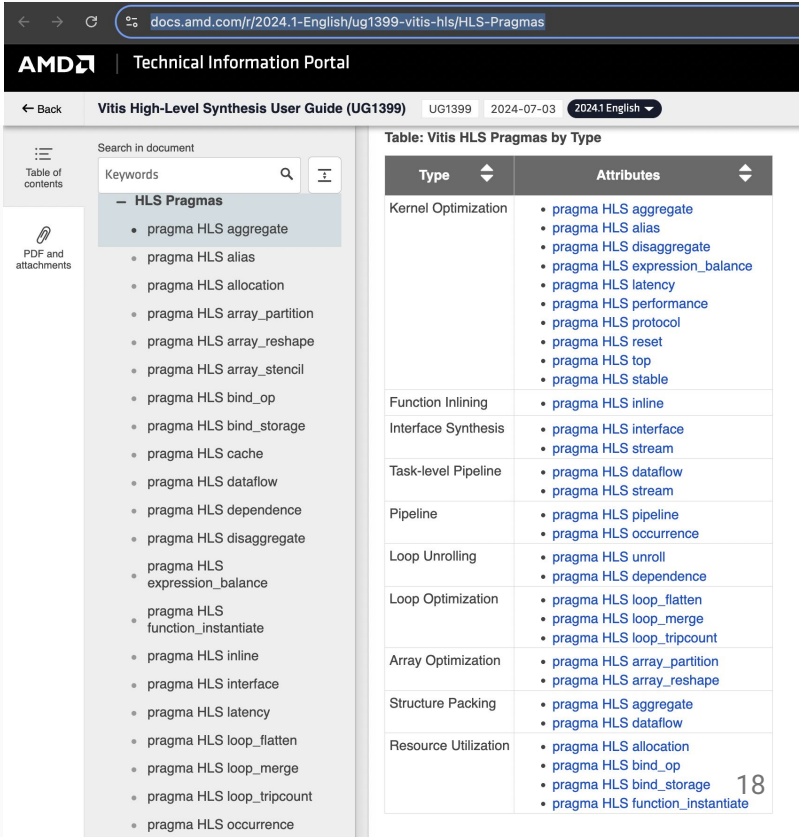
# Arrays

- Small arrays can be implemented as **local registers** but **bigger arrays must be implemented in memory** (BRAM/URAM/etc), which implies much longer latency and uses precious resources
- Can use tools and pragmas to **reshape and organize arrays** to minimize memory bottlenecks and latency issues

```
char my_array[2][4] = {"hip", "hop"};
```

# Sidebar: Pragmas


- **Pragmas** and you will will be **besties**!
- Array optimization and aggregate/disaggregate pragmas optimize memory access
- Unrolling, pipelining, dataflow pragmas control the exploitation of parallelism
- Resource utilization pragmas more finely control what resources are used
- Other pragmas are worth exploring but not discussed here



# Back to our diamond function: **pipeline** pragma

```
#pragma HLS pipeline II=<int> off rewind style=<value>
```

Where:

`II=<int>`

Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval.

`off`

Optional keyword. Turns off pipeline for a specific loop or function. This can be used to disable pipelining for a specific loop when `config_compile -pipeline_loops` is used to globally pipeline loops.

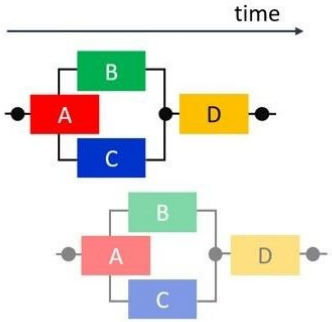
`rewind`

Optional keyword. Enables rewinding as described in Rewinding Pipelined Loops for Performance. This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:

- Is considered as initialization.
- Is executed only once in the pipeline.
- Cannot contain any conditional operations (if-else).

★ **Tip:** This feature is only supported for pipelined loops; it is not supported for pipelined functions.

```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
   data_t c1[N], c2[N], c3[N], c4[N];
   #pragma HLS dataflow
   A(vecIn, c1, c2);
   B(c1, c3);
   C(c2, c4);
   D(c3, c4, vecOut);
}
```



# Back to our diamond function: **pipeline** pragma

```
#pragma HLS pipeline II=<int> off rewind style=<value>
```

Where:

`II=<int>`

Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval.

**Target II (not guaranteed to be met)**

`off`

Optional keyword. Turns off pipeline for a specific loop or function. This can be used to disable pipelining for a specific loop when `config_compile -pipeline_loops` is used to globally pipeline loops.

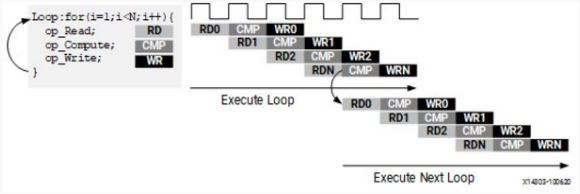
`rewind`

Optional keyword. Enables rewinding as described in Rewinding Pipelined Loops for Performance. This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:

- Is considered as initialization.
- Is executed only once in the pipeline.
- Cannot contain any conditional operations (if-else).

★ **Tip:** This feature is only supported for pipelined loops; it is not supported for pipelined functions.

**Allows pipelining between "events" or entries into the loop!**



# Back to our for loops: **unroll** pragma

```
for(int i = 0; i < X; i++) {
  pragma HLS unroll factor=2
  a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following code where the `break` construct is used to ensure the functionality remains the same, and the loop exits at the appropriate point.

```
for(int i = 0; i < X; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= X) break;
  a[i+1] = b[i+1] + c[i+1];
}
```

You can unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations can also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). Using the UNROLL pragma you can unroll loops to increase data access and throughput.

The UNROLL pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor *N*, to create *N* copies of the loop body and reduce the loop iterations accordingly.

★ **Tip:** To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Partial loop unrolling does not require *N* to be an integer factor of the maximum loop iteration count. The Vitis HLS tool adds an exit check to ensure that partially unrolled loops are functionally identical to the original loop. For example, given the following code:

**Of course this only works because ith and (i+1)th elements are independent**

# Back to our for loops: **unroll** pragma

```
#pragma HLS unroll factor=<N> skip_exit_check off=true
```

`factor=<N>`

Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If `factor=` is not specified, the loop is fully unrolled.

`skip_exit_check`

Optional keyword that applies only if partial unrolling is specified with `factor=`. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:

- Fixed bounds

  No exit condition check is performed if the iteration count is a multiple of the factor.

  If the iteration count is *not* an integer multiple of the factor, the tool:

  - Prevents unrolling.
  - Issues a warning that the exit check must be performed to proceed.
- Variable bounds

  The exit condition check is removed. You must ensure that:

  - The variable bounds is an integer multiple of the factor.
  - No exit check is in fact required.

`off=true`

Disable unroll for the specified loop.

**Fully unrolling is not always possible, need to not have extremely large loops or loops with unknown bounds at compile time!**

# Back to arrays: **array_partition** pragma

```
char my_array[2][9] = {"of",
"sunshine"};
```

```
#pragma HLS array_partition variable=<name> \
type=<type>  factor=<int>  dim=<int> off=true
```

**Splits the arrays into multiple smaller arrays: multiple arrays increases the bandwidth that one can access the data**

factor = 3

| array element | new array: block | new array: cyclic |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 0 |
| 4 | 2 | 1 |
| 5 | 2 | 2 |

`variable=<name>`
A required argument that specifies the array variable to be partitioned.

`type=<type>`
Optionally specifies the partition type. The default type is `complete`. The following types are supported:

`cyclic`
Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if `factor=3` is used:

- Element 0 is assigned to the first new array
- Element 1 is assigned to the second new array.
- Element 2 is assigned to the third new array.
- Element 3 is assigned to the first new array again.

`block`
Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.

`complete`
Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default `<type>`.

`factor=<int>`
Specifies the number of smaller arrays that are to be created.

**!! Important:** For complete type partitioning, the factor is not specified. For block and cyclic partitioning, the `factor=` is required.

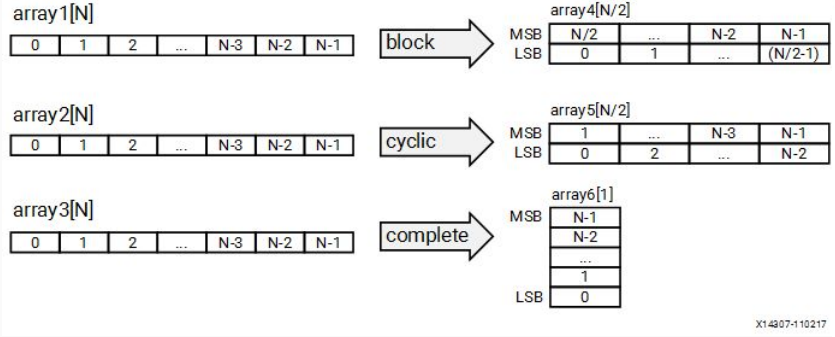
**Often complete partitioning is used for small arrays (i.e. each element becomes its own array) so that all the values can be accessed in parallel**

# Back to arrays: **array_reshape** pragma


**Reshapes the array into one with fewer, wider elements. Can decrease the number of memories used, and more data is provided at once**

`factor=<int>`

Specifies the amount to divide the current array by (or the number of temporary arrays to create). A factor of 2 splits the array in half, while doubling the bit-width. A factor of 3 divides the array into three, with triple the bit-width.




`type=<type>`

Optionally specifies the partition type. The default type is `complete`. The following types are supported:

`cyclic`

Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if `factor=3` is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.

`block`

Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into `<N>` equal blocks where `<N>` is the integer defined by `factor=`, and then combines the `<N>` blocks into a single array with `word-width*N`.

`complete`

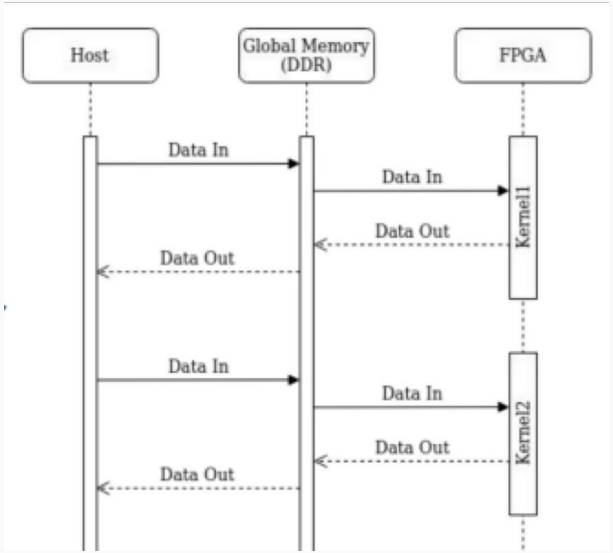
Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with `N*M` bits). This is the default type of array reshaping.

```
#pragma HLS array_reshape variable=<name> type=<type>  factor=<int>  dim=<int> off=true
```

# Interfaces


**Interfaces are complex, as the FPGA communicates through PCIe bus**
**But generally, you put the data from CPU memory into FPGA DDR, work on it, put the results in DDR and read it back**

- General Rules of thumbs:
  - DDR read is slow, typically read once and operate many times
  - Don’t jump around, process data sequentially
  - If you need data later, save it in local memory

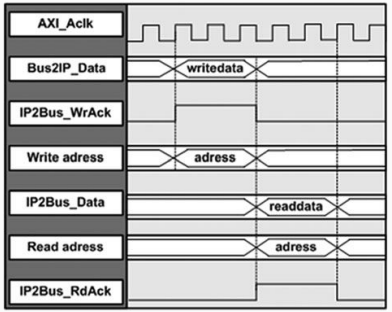


# Interface pragma

**Interfaces pragmas are complex**

**But typical rule of thumb, define your inputs and outputs… and maybe control signals**

(Behind the scene, we are using the AXI protocol, which has many configurable knobs - Haider doesn't know even half of them despite working with them for 2 years)




**AXI lite communication pattern**

```
// TOP LEVEL - TRANSPOSE
void transpose(int* input, int* output) {
    #pragma HLS INTERFACE mode=m_axi port=input offset=slave bundle=gmem0
    #pragma HLS INTERFACE mode=m_axi port=output offset=slave bundle=gmem1

    #pragma HLS INTERFACE mode=s_axilite port=input bundle=control
    #pragma HLS INTERFACE mode=s_axilite port=output bundle=control
    #pragma HLS INTERFACE mode=s_axilite port=return bundle=control

    #pragma HLS dataflow
```

**Full AXI ports for data input**

**Control signal for the ports, done automatically, so unless you know what you are doing you can ignore these**

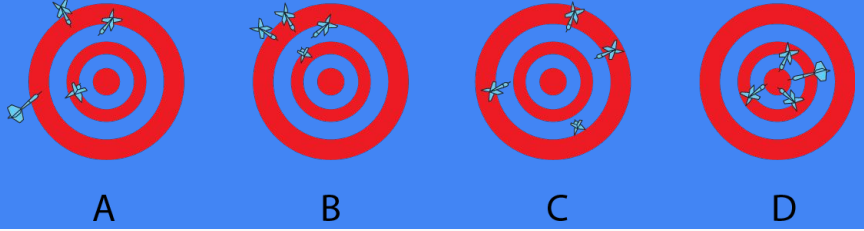
# Streaming pragma

- To ensure not jumping around, typically we always read from DDR into a streaming, and process on stream
  - Stream is a FIFO, where once the data is used, its gone
  - Also allows optimization and parallelization from vitis, as we don't depend on slow DDR
  - Stream depths are important, since if they are full, the HW will stall unless something empties it

```
extern "C" {
    void example(int *inx, int *outx, int alpha) {
    // AXI master interface
    #pragma HLS INTERFACE m_axi port = inx offset = slave bundle = gmem
    #pragma HLS INTERFACE m_axi port = outx offset = slave bundle = gmem
    // AXI slave interface
    #pragma HLS INTERFACE s_axilite port = inx bundle = control
    #pragma HLS INTERFACE s_axilite port = outx bundle = control
    #pragma HLS INTERFACE s_axilite port = alpha bundle = control
    #pragma HLS INTERFACE s_axilite port = return bundle = control

        hls::stream<int> inFifo("inFifo");
        #pragma HLS STREAM variable=inFifo depth=128
        hls::stream<int> outFifo("outFifo");
        #pragma HLS STREAM variable=inFifo depth=128

        // Dataflow enables task level pipelining, allowing functions and loops
        // to execute concurrently. Used to minimize interval.
        #pragma HLS DATAFLOW
        // Read data
        read_data(inx, inFifo);
        // Do computation with the acquired data
        compute(inFifo, outFifo, alpha);
        // Write data
        write_data(outx, outFifo);
        return;
    }
}
```

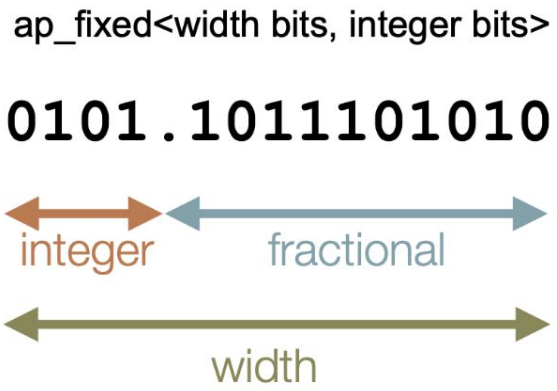
```
// Read data function : Read Data from Global Memory
void read_data(DTYPE *inx, hls::stream<int> &inFifo) {
read_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
    read_loop_jj:
        for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
            #pragma HLS PIPELINE II=1
            inFifo << inx[WORD_PER_ROW * i + jj];
            ;
        }
    }
}

// Write data function : Write Results to Global Memory
void write_data(DTYPE *outx, hls::stream<int> &outFifo) {
write_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
    write_loop_jj:
        for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
            #pragma HLS PIPELINE II=1
            outFifo >> outx[WORD_PER_ROW * i + jj];
        }
    }
}
```

# Precision and data types

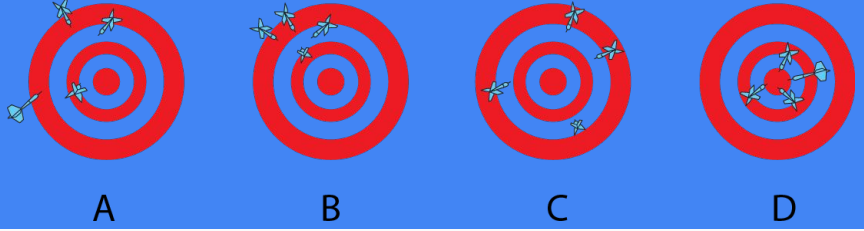



- Can use standard data types (float, int, unsigned, etc) but these use a fixed number of bits, sometimes not optimized for your work
- Can instead use Arbitrary Precision (AP) data types to determine the number of bits needed, the range, rounding, overflow, the sign of the

| Type | Description | Numerical Range | Quantum |
|---|---|---|---|
| *ac_int*<W, false> | unsigned integer | 0 to $2^{W}$- 1 | 1 |
| *ac_int*<W, true> | signed integer | $-2^{W-1}$ to $2^{W-1}$- 1 | 1 |
| *ac_fixed*<W, I, false> | unsigned fixed-point | 0 to (1 - $2^{-W}$) $2^{I}$ | $2^{I-W}$ |
| *ac_fixed*<W, I, true> | signed fixed-point | (-0.5) $2^{I}$ to (0.5 - $2^{-W}$) $2^{I}$ | $2^{I-W}$ |



# Precision and data types

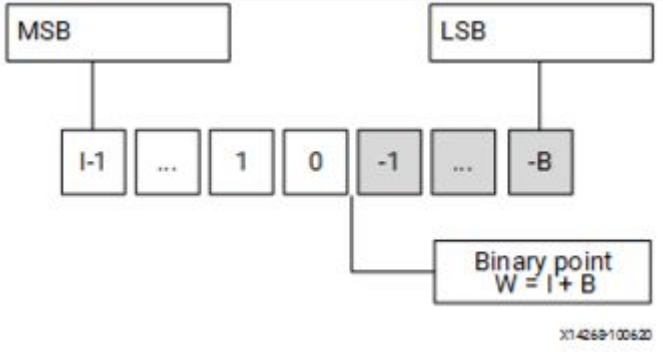



**Integer Data Type**

ap_[u]int<W> (1024 bits)

Can be extended to 4K bits wide as explained in C++ Arbitrary Precision Integer Types.

ap_[u]fixed<W,I,Q,O,N>




| | |
|---|---|
| Q | Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result. |

| ap_fixed Types | Description |
|---|---|
| AP_RND | Round to plus infinity |
| AP_RND_ZERO | Round to zero |
| AP_RND_MIN_INF | Round to minus infinity |
| AP_RND_INF | Round to infinity |
| AP_RND_CONV | Convergent rounding |
| AP_TRN | Truncation to minus infinity (default) |
| AP_TRN_ZERO | Truncation to zero |

| | |
|---|---|
| O | Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result. |

| ap_fixed Types | Description |
|---|---|
| AP_SAT [1] | Saturation |
| AP_SAT_ZERO [1] | Saturation to zero |
| AP_SAT_SYM [1] | Symmetrical saturation |
| AP_WRAP | Wrap around (default) |
| AP_WRAP_SM | Sign magnitude wrap around |

| | |
|---|---|
| W | Word length in bits |
| I | The number of bits used to represent the integer value, that is, the number of integer bits to the *left* of the binary point, including the sign bit.<br><br>When I is negative, as shown in the example below, it represents the number of *implicit* sign bits (for signed representation), or the number of *implicit* zero bits (for unsigned representation) to the *right* of the binary point. For example, |

```
ap_fixed<2, 0> a = -0.5;     // a can be -0.5,

ap_ufixed<1, 0> x = 0.5;     // 1-bit representation. x can be 0 or 0.5
ap_ufixed<1, -1> y = 0.25;   // 1-bit representation. y can be 0 or 0.25
const ap_fixed<1, -7> z = 1.0/256;  // 1-bit representation for z = 2^-8
```

**Important to think in advance about sign and range of numbers and the precision you need!**

# Conclusion: None of this is easy, but it is fun!


**Jahred's favorite HLS example - these two pieces of code from Vitis examples give the same output, but one is much more optimal than the other. Why?**

```
dout_t array_mem_bottleneck(din_t mem[N]) {

    dout_t sum = 0;
    int i;

SUM_LOOP:
    for (i = 2; i < N; ++i)
        sum += mem[i] + mem[i - 1] + mem[i - 2];

    return sum;
}
```

```
dout_t mem_bottleneck_resolved(din_t mem[N]) {

    din_t tmp0, tmp1, tmp2;
    dout_t sum = 0;
    int i;

    tmp0 = mem[0];
    tmp1 = mem[1];
SUM_LOOP:
    for (i = 2; i < N; i++) {
        tmp2 = mem[i];
        sum += tmp2 + tmp1 + tmp0;
        tmp0 = tmp1;
        tmp1 = tmp2;
    }

    return sum;
}
```