Bridge of Life Education

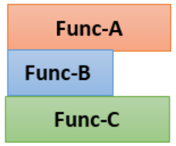
Dataflow

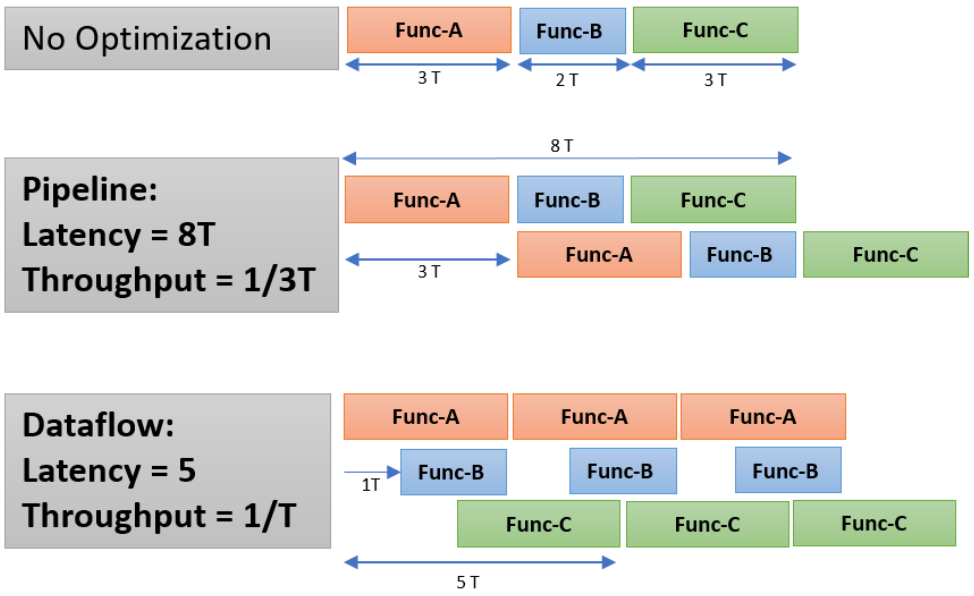
Dataflow

Characteristics:

1. Sequence of tasks runs concurrently.
2. Sequence of tasks runs without waiting for the previous tasks to complete.
3. The start of the task is subject to the availability of the data coming from the previous task.
4. The sequence of tasks communicates by a channel. The channel structure can be FIFO or Memory.

```
Void top(a,b,c,d) {
  Func_A(a,b,i1);     3T
  Func_B(c,i1,i2);    2T
  Func_C(i2,d);       3T
}
```

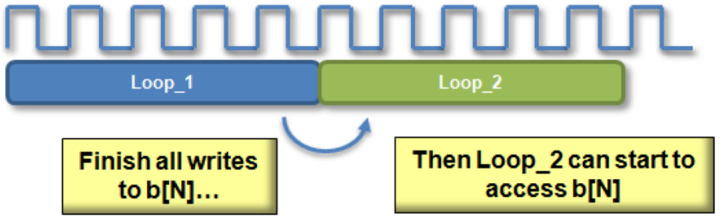




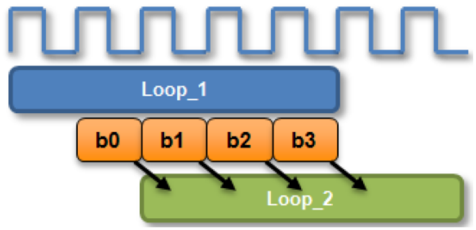
Dataflow : Ideal for streaming arrays & multi-rate functions

- **Arrays are passed as single entities by default**
  - This example uses loops but the same principle applies to functions

```
int a[N], b[N], c[N];

Loop_1: for (i=0;i<=N-1;i++) {    Loop_1
    b[i] = a[i] + in1;
}

Loop_2: for (i=0;i<=N-1;i++) {    Loop_2
     c[i] = b[i] * in2;
}
```



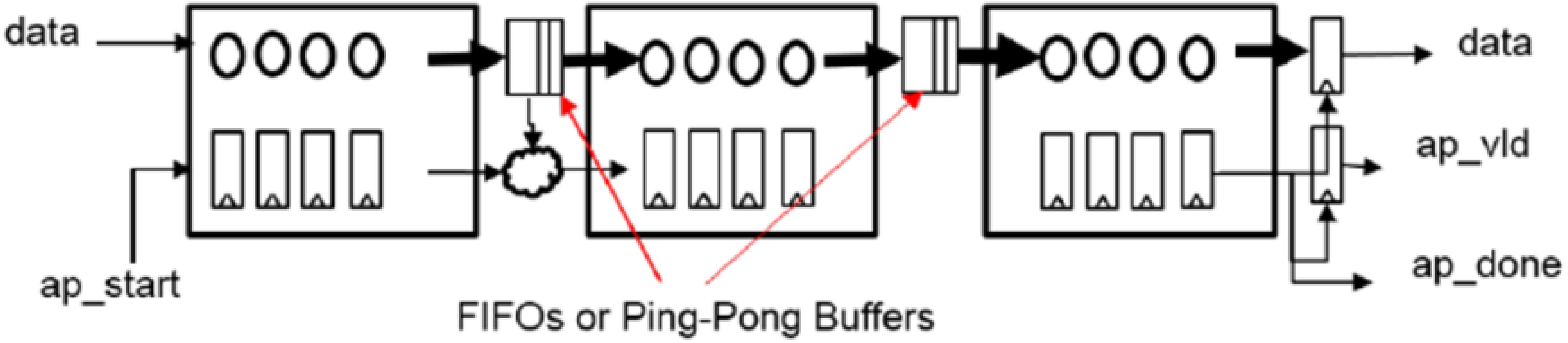
- **Dataflow pipelining allows loop_2 to start when data is ready**
  - The throughput is improved
  - Loops will operate in parallel
    - If dependencies allow



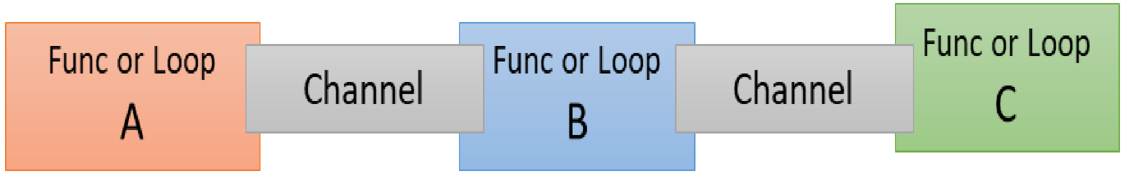
- **Multi-Rate Functions**
  - Dataflow buffers data when one function or loop consumes or produces data at different rate from others
- **IO flow support**
  - To take maximum advantage of dataflow in streaming designs, the IO interfaces at both ends of the datapath should be streaming/handshake types (ap_hs or ap_fifo)

Dataflow Structure

- HLS inserts channel between blocks of code
  - The blocks can be functions or loops
  - Data buffer: FIFO or Ping-pong buffer
    - For arrays, the channel uses memory to buffer the samples
    - For scalar, the channel is a register with hand-shakes
  - Handshake signals
- Not limited to a chain of process, Used on any DAG structure
- centrally-controlled pipeline by FSM v.s. Distributed handshaking architecture v.s. (reduce fanout of control signals)

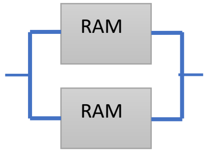


Dataflow Channel - FIFO or Ping-Pong Buffer



FIFO

- For scalar, pointer, reference parameters
- Access in sequential order (depth 2, use –fifo_depth)



Ping pong buffer

Ping-Pong Buffer

- For array
  - Two block RAMs
  - Memory size = Maximum number of consumer or producer elements (default)

- **How to use FIFO for Array**
  - At top function interface: set as ap_fifo, axis, ap_hs
  - At inside function: use STREAM

https://www.xilinx.com/developer/articles/task-level-parallelism-and-pipelining-in-hls.html

Dataflow Canonical Forms

Dataflow Canonical Form - Function

```
static void read_input( unsigned int* in,  hls::stream<unsigned int>& inStream,  int size) {
mem_rd: // HLS automatically apply pipeline to this loop
    for (int i = 0; i < size; i++) {
        #pragma PIPELINE II = 1
        inStream << in[i];              // Blocking write command to inStream } }
```

```
static void compute_add(hls::stream<unsigned int>& inStream,  hls::stream<unsigned int>&
outStream,  int inc,  int size) {
execute:         // Auto-pipeline is going to apply pipeline to this loop
    for (int i = 0; i < size; i++) {
        #pragma PIPELINE II = 1
        outStream << (inStream.read() + inc); }}
```

```
static void write_result(unsigned int* out,  hls::stream<unsigned int>& outStream,
                         int size) {
mem_wr:  // Auto-pipeline is going to apply pipeline to this loop
    for (int i = 0; i < size; i++) {
         #pragma PIPELINE II = 1
         out[i] = outStream.read();   } }
```

```
void adder(unsigned int* in, unsigned int* out, int inc, int size) {
    static hls::stream<unsigned int> inStream("input_stream");
    static hls::stream<unsigned int> outStream("output_stream");
#pragma HLS STREAM variable = inStream depth = 32
#pragma HLS STREAM variable = outStream depth = 32
#pragma HLS dataflow
    read_input(in, inStream, size);
    compute_add(inStream, outStream, inc, size);
    write_result(out, outStream, size);} }
```

- The function can not be inlined
- #pragma HLS STREAM – FIFOs are used
- #pragma HLS DATAFLOW

```
 _____________
|             |<----- Input Vector from Global Memory
|  read_input |       __
|_____________|----->|  |
 _____________       |  | inStream
|             |<-----|__|
| compute_add |       __
|_____________|----->|  |
 _____________       |  | outStream
|              |<----|__|
| write_result |
|______________|-----> Output result to Global Memory
```


https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Canonical-Forms

Dataflow Canonical Form - inside a Loop Body

Requirements:

- Initial value declared in the loop header and set to 0.
- The loop condition is a positive numerical constant or constant function argument.
- Increment by 1
- Dataflow pragma needs to be inside the loop.
- Should have a single loop counter.
- The inner loop needs to pipeline

```
wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {

  #pragma HLS DATAFLOW
  wr_buf_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
    wr_buf_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
      #pragma HLS PIPELINE
      // should burst TILE_WIDTH in WORD beat
      outFifo >> tile[m][n];
    }  }

  wr_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
    wr_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
      #pragma HLS PIPELINE
      outx[TILE_HEIGHT*TILE_PER_ROW*TILE_WIDTH*i
          +TILE_PER_ROW*TILE_WIDTH*m+TILE_WIDTH*j+n] = tile[m][n];
    } }
}
```

Check if it meets all the requirements

Dataflow does not propagate through hierarchy

If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the DATAFLOW optimization to the loop, the sub-function, or **inline** the sub-function.

Use ap_ctrl_none for Dataflow

```
#pragma HLS interface ap_ctrl_none port=return
```

- All processes specified ap_ctrl_none
- All processes executed or stalled based on the availability of data in the FIFO, no synchronization on the block level.
- **Processes are executed different rate, different latency**

Processes Execute at Different Rate – Rate Matching

- Faster process distributed work to several slower ones

```
void region(...) {
   #pragma HLS dataflow
   #pragma HLS interface ap_ctrl_none port=return
   hls::stream<int> outStream1, outStream2;

   demux(inStream, outStream1, outStream2);
   worker1(outStream1, ...);
   worker2(outStream2, ....);
```

demux: II = 1
worker1: II = 2
worker2: II = 1

Process at different Latency - Latency Matching
Estimate HLS Stream Depth

- Paths with different latency re-converge to a process

```
hls::Duplicate(src,src0,src1);
hls::GaussianBlur<5,5>(src1, blurr1);
hls::GaussianBlur<3,3>(src0, blurr0);
hls::AbsDiff(blurr0,blurr1,dst);

       /---\                                /-----\
       | D | -> src1 -> [5x5] -> blurr1 -> | ABS |
src -> | U |                                |     | -> dst
       | P | -> src0 -> [3x3] -> blurr0 -> | DIFF|
       \---/                                \-----/
```

  - Kernel of NxN would have N lines of latency
  - FIFO with 2 lines of latency on the 3x3 path to balance 5x5 paths

Pragma related to Dataflow

- **#pragma HLS dataflow [disable_start_propagation]**
  - disable_start_propagation: Optionally disables the creation of a start FIFO used to propagate a start token to an internal process. Such FIFOs can sometimes be a bottleneck for performance.
- **#pragma HLS stream variable=<variable> depth=<int> off**
  - variable=<variable>: Specifies the name of the array to implement as a streaming interface.
  - depth=<int>:
    - For array streaming in DATAFLOW channels.
    - Default, the depth of the FIFO is the same size as the array specified in the C code.
    - If producer and consumer at the same throughput, FIFO can be reduced to 1
  - off: Disables streaming data. Relevant only for array streaming in dataflow channels. (override the config_dataflow –default_channel fifo)

Latency Difference

```
void adder(unsigned int *in, unsigned int *out, int inc, int size)
{
   hls::stream<unsigned int> inStream;
   hls::stream<unsigned int> outStream;
#pragma HLS STREAM variable=inStream  depth=32
#pragma HLS STREAM variable=outStream depth=32

#pragma HLS dataflow

   mem_rd: for (int i = 0 ; i < size ; i++){
#pragma HLS LOOP_TRIPCOUNT min=4096 max=4096
      inStream << in[i];
   }

   execute: for (int j = 0 ; j < size ; j++){
#pragma HLS LOOP_TRIPCOUNT min=4096 max=4096
      outStream << (inStream.read() + inc);
   }
   mem_wr: for (int k = 0 ; k < size ; k++) {
#pragma HLS LOOP_TRIPCOUNT min=4096 max=4096
      out[k] = outStream.read();
   }
}
```

No DATAFLOW

Latency Information (clock cycles)

Compute Unit	Kernel Name	Module Name	Start Interval	Best Case	Avg Case	Worst Case
adder_1	adder	adder	12309	12308	12308	12308

With DATAFLOW

Latency Information (clock cycles)

Compute Unit	Kernel Name	Module Name	Start Interval	Best Case	Avg Case	Worst Case
adder_1	adder	Loop_mem_rd_proc235	4105	4105	4105	4105
adder_1	adder	Loop_execute_proc	4098	4098	4098	4098
adder_1	adder	Loop_mem_wr_proc	4104	4104	4104	4104
adder_1	adder	adder	4106	4112	4112	4112

https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/rfo1504034311160.html

Rules for Dataflow

Rules for Dataflow

- Rules for variables
  1. Use a local, non-static scalar or array/pointer variables, or
  2. local static stream variable.
  3. Declared inside the dataflow region,
     - inside the function body (for dataflow in a function) or
     - loop body (for dataflow inside a loop)
- A sequence of function calls forward (no feedback)
- Variables can have only one reading process and one writing process
- Function return type must be void
- No loop carried dependencies

Limitations

- Reading from function inputs or writing to function outputs in the middle of the dataflow region
- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution tasks
- Loop with multiple exit conditions

Use Dataflow viewer in the Analysis perspective

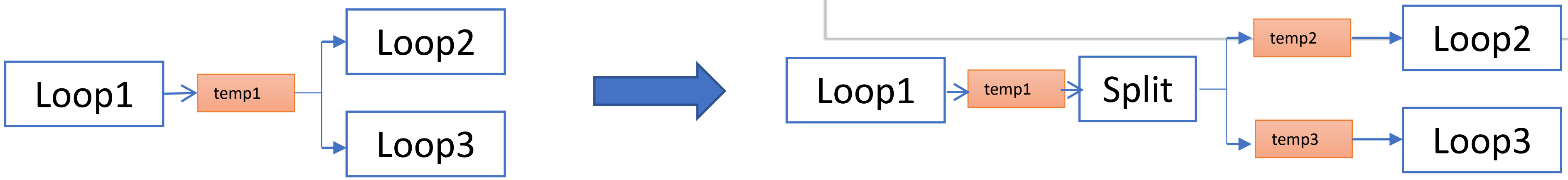
Single-Producer-Consumer Violation

Single-Producer-Consumer Violations

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

int temp1 [N];
Loop1: for(int i = 0; i <N; i++) {
        temp1 [i] = data_in[i] * scale;
}
Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] =temp1 [j] * 123;
}
Loop3: for(int k = 0; k < N; K++) {
        data_out2[j] = temp1 [k] * 456;
}

}
```

Solution

```
void Split (in[N], out1[N], out [N]) {
// Duplicated data
   L1:for (int i=1; i<N; i++) {
      out1 [i] = in [i];
      out2 [i] = in [i];
   }
}
void foo(int data_in[N], int scale, int data_out1 [N], int data_out2[N]) {

   int temp1[N], temp2[N] . temp3[N];
   Loop1: for (int i = 0; i < N; i++) {
      temp1[i] = data_in[i] * scale;
   }
   Split (temp1, temp2, temp3);
   Loop2: for(int j = 0; j < N; j++) {
      data_out1[j] = temp2[j] * 123;
   }
   Loop3: for(int k = 0; k < N; k++) {
      data_out2[j] = temp3[k] * 456;
   }
}
```



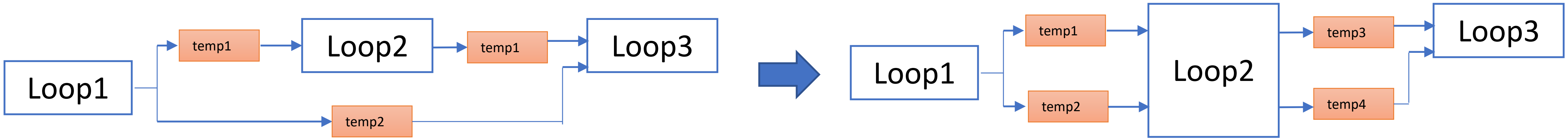
Bypassing Tasks

Bypassing Tasks

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

   int temp1(N), temp2(N). temp3(N);
   Loop1: for(int i = 0; i < N; i++) {
      temp1[i] = data_in[i] * scale;
      temp2[i] = data_in[i] >> scale;
   }
   Loop2: for(int j = 0; j < N; j++) {
      temp3[j] = temp1[j] + 123;
   }
   Loop3: for(int k = 0; k < N; k++) {
      data_out[j] = temp2[k] + temp3[k];
   }
}
```

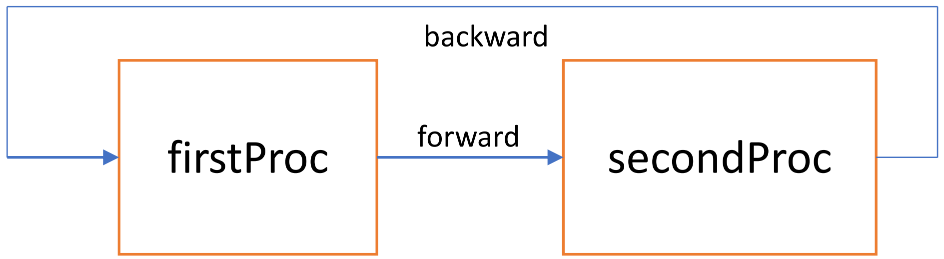
Solution

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

   int temp1[N], temp2[N]. temp3[N], temp4[N];
   Loop1: for(int i = 0; i < N; i++) {
      temp1[i] = data_in[i] * scale;
      temp2[i] - data_in[i] >> scale;
   }
   Loop2: for(int j = 0; j < N; j++) {
      temp3[j] = temp1[j] + 123;
      temp4[j] = temp2[j];
   }
   Loop3: for(int k = 0; k < N; k++) {
      data_out(j] = temp4[k] + temp3[k];
   }
}
```



Feedback between Tasks

- Feedback between tasks is not recommended . When detected, it issues a warning. Might not perform the DATAFLOW optimization.
- DATAFLOW can support feedback when used with hls::streams.




```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

void firstProc(hls::stream<int> &forwardOUT, hls::stream<int> &backwardIN) {
  static bool first = true;
  int fromSecond;

  //Initialize stream
  if (first)
    fromSecond = 10; // Initial stream value
  else
    //Read from stream
    fromSecond = backwardIN.read(); //Feedback value
  first = false;

  //Write to stream
  forwardOUT.write(fromSecond*2);
}

void secondProc(hls::stream<int> &forwardIN, hls::stream<int> &backwardOUT) {
  backwardOUT.write(forwardIN.read() + 1);
}

void top(...) {
#pragma HLS dataflow
  hls::stream<int> forward, backward;
  firstProc(forward, backward);
  secondProc(forward, backward);
}
```


https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Dataflow-Optimization-Limitations

Conditional Execution of Tasks

Ensure each is executed in all cases. Both loops are always executed, and data always flows from one loop to the next.

```
void foo(int data_in1[N], int data_out[N], int sel) {

 int temp1[N], temp2[N];

 if (sel) {
 Loop1: for(int i = 0; i < N; i++) {
 temp1[i] = data_in[i] * 123;
 temp2[i] = data_in[i];
 }
 } else {
 Loop2: for(int j = 0; j < N; j++) {
 temp1[j] = data_in[j] * 321;
 temp2[j] = data_in[j];
 }
 }
 Loop3: for(int k = 0; k < N; k++) {
 data_out[k] = temp1[k] * temp2[k];
 }
}
```

```
void foo(int data_in[N], int data_out[N], int sel) {

 int temp1[N], temp2[N];

 Loop1: for(int i = 0; i < N; i++) {
 if (sel) {
 temp1[i] = data_in[i] * 123;
 } else {
 temp1[i] = data_in[i] * 321;
 }
 }
 Loop2: for(int j = 0; j < N; j++) {
 temp2[j] = data_in[j];
 }
 Loop3: for(int k = 0; k < N; k++) {
 data_out[k] = temp1[k] * temp2[k];
 }
}
```

Loops with Multiple Exit Conditions

Loop2 has three exit conditions:

- An exit defined by the value of N; the loop will exit when k>=N.
- An exit defined by the break statement.
- An exit defined by the continue statement.

=> **Break, Continue statements can not be used in loop**

```
void multi_exit(din_t data_in[N], dsc_t scale,
                    dsel_t select, dout_t data_out[N]) {

  dout_t temp1[N], temp2[N];
  int i,k;
  Loop1: for(i = 0; i < N; i++) {
    temp1[i] = data_in[i] * scale;
    temp2[i] = data_in[i] >> scale;
  }
  Loop2: for(k = 0; k < N; k++) {
    switch(select) {
      case 0: data_out[k] = temp1[k] + temp2[k];
      case 1: continue;
      default: break;
    }
  }
}
```

hls:stream for Dataflow

Stream for Dataflow : hls:stream<>

- Include <hls_stream.h>
- **hls::stream<Type, Depth>**
  - Type:
    - C++ native data type
    - HLS arbitrary precision type, e.g. ap_int<>
    - User-defined struct containing above types
  - Depth: depth of FIFO for co-simulation verification
- Used for top-level function arguments, and between functions
- Top interface can be
  - FIFO interface: ap_fifo (default) - support non-blocking behavior
  - Handshake interface: ap_hs
  - AXI4-Stream : axis
- Inside function – FIFO with depth = 2 (#pragma HLS STREAM depth = <int>), note: depth specify actual resource allocation.
- Use passed-by-reference to pass streams into and out of functions
- Only in C++ based designs

Stream Example

> **Create using hls::stream or define the hls namespace**

```
#include <ap_int.h>
#include <hls_stream.h>

typedef ap_uint<128> uint128_t;                // 128-bit user defined type

hls::stream<uint128_t> my_wide_stream;    // A stream declaration
```

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;                         // Use hls namespace

typedef ap_uint<128> uint128_t;           // 128-bit user defined type

stream<uint128_t> my_wide_stream;     // hls:: no longer required
```

> **Blocking and Non-Block accesses supported**

```
// Blocking Write
hls::stream<int> my_stream;

int src_var = 42;
my_stream.write(src_var);
// OR use:  my_stream << src_var;
```

```
// Blocking Read
hls::stream<int> my_stream;

int dst_var;
my_stream.read(dst_var);
// OR use: dst_var = my_stream.read();
```

```
hls::stream<int> my_stream;

int src_var = 42;
bool stream_full;

// Non-Blocking Write
if (my_stream.write_nb(src_var)) {
     // Perform standard operations
} else {
     // Write did not happen
}

// Full test
stream_full = my_stream.full();
```

```
hls::stream<int> my_stream;

int dst_var;
bool stream_empty;

// Non-Blocking Read
if (my_stream.read_nb(dst_var)) {
     // Perform standard operations
} else {
      // Read did not happen
}

// Empty test
fifo_empty = my_stream.empty();
```

Stream arrays and structs are not supported for RTL simulation at this time: must be verified manually.

e.g. hls::stream<uint8_t> chan[4])

C Modeling and C/RTL Co-simulation

- C-simulation : models as infinite queue. No need to specify depth
- C/RTL Co-simulation with ap_ctrl_none
  - combinational designs
  - pipelined design with task interval of 1
  - designs with array streaming or hls_stream or AXI4 stream ports.

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3) designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

- Co-simulation: depth sufficient to hold the test-bench data
  - Stream read by the top-level design must be pre-loaded with data in the C++ testbench before calling the function
  - FIFO depth determined by co-sim with the histogram of the occupation of each FIFO/PIPO buffer

```
#pragma HLS STREAM variable= xx depth = yy
set_directive_stream –depth yy
```

- C/RTL does not support structures or classes containing hls::stream<> members in the top-level interface. If struct of streams are used for synthesis, the design must be verified using an external RTL simulator and user-created HDL test bench

```
typedef struct {
   hls::stream<uint8_t> a;
   hls::stream<uint16_t> b;
} strm_strct_t;

void dut_top(strm_strct_t indata, strm_strct_t outdata) { }
```


https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/C/RTL-Co-Simulation-Support
https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Stream-Library

Co-simulation with Stream

```
int main(void) {
    hls::stream<uint4> din, dout;
    hls::stream<ctrl_type> ctrl;
    uint4 output;
    int pass = 1;

    for (int i = 0; i < len; i++)   din.write(i);
    ctrl.write(len);

    top(din, dout, ctrl);

    for (int i = 0; i < len; i++) {
        output = dout.read();
        cout << "Output: " << output << "\tExpect: " <<
uint4(i * 13) << endl;
        if (output != uint4(i * 13)) pass = 0;
    }
}
```

```
void top(
        hls::stream<uint4>& din,
        hls::stream<uint4>& dout,
         hls::stream<ctrl_type>& ctrl) {

#pragma HLS dataflow
    hls::stream<uint4> data_int;
    hls::stream<ctrl_type> ctrl_int;

    BLOCK0(din, data_int, ctrl, ctrl_int);
    BLOCK1(data_int, dout, ctrl_int);
}
```