



Bridge of Life
Education

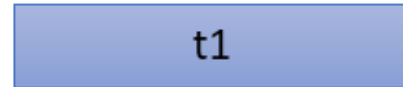
PIPELINE

Topics

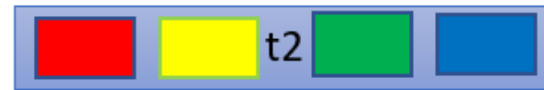
- Latency v.s. Throughput
- Loop & Function Pipeline
- Loop Rewind & Flush
- Loop Latency – Merge & Flatten
- Pipeline limitation – II
- Unroll

Pipeline Latency v.s. Throughput

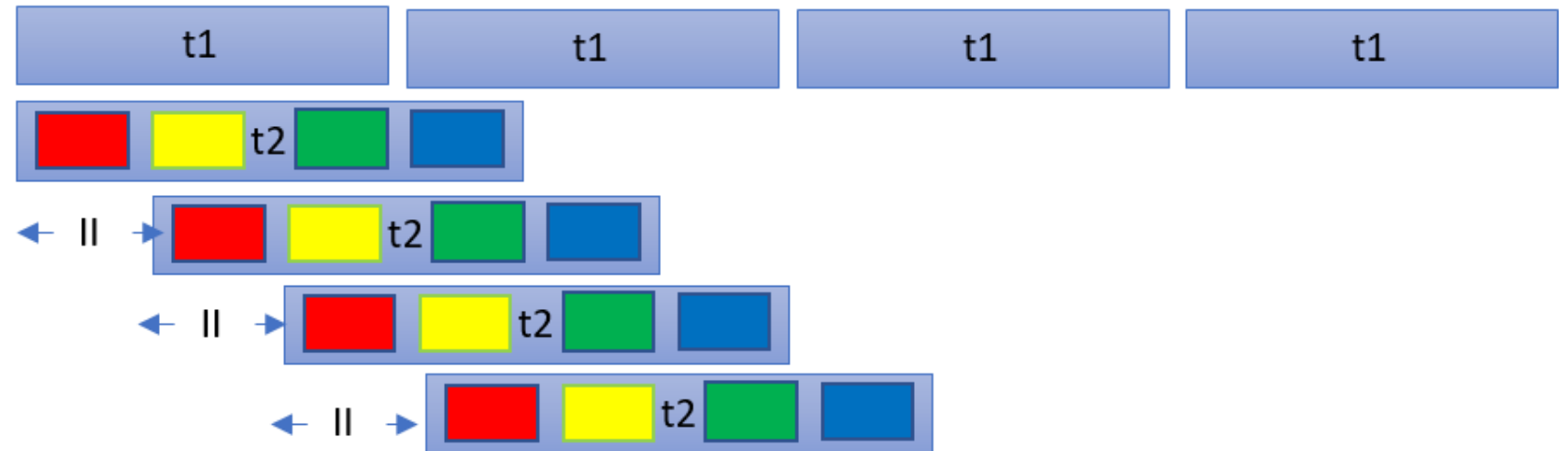
A function, latency t_1



Decompose a function into 4 steps, latency t_2

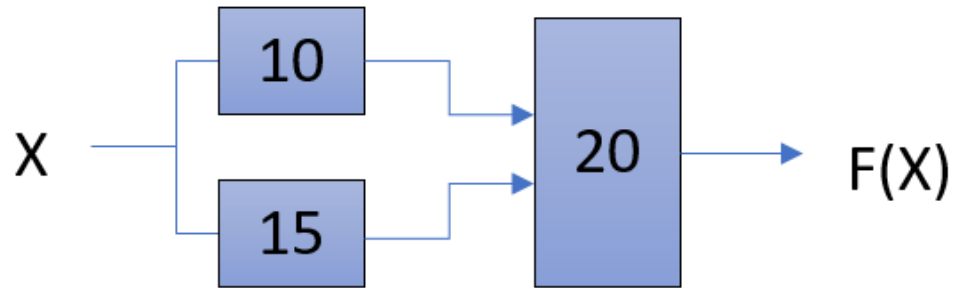


Compare the latency to execute 4 iterations of non-pipelined function ($4 \cdot t_1$) v.s. pipelined functions ($t_2 + 3 \text{ II}$)

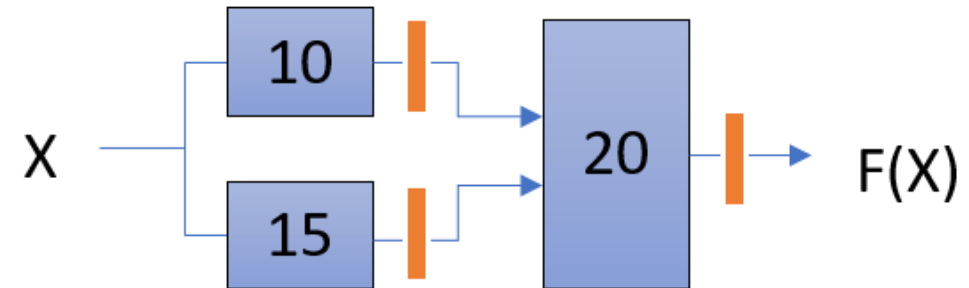


II: Initiation Interval

Pipeline May Take Longer Latency



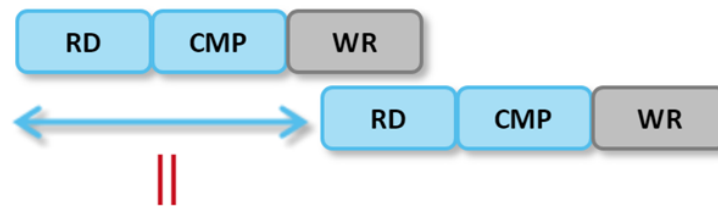
Latency = 35
Throughput = $1/35$



T_{clk} (clock period) = 20
Latency = $2 * T_{clk} = 40$
Throughput = $1/20$

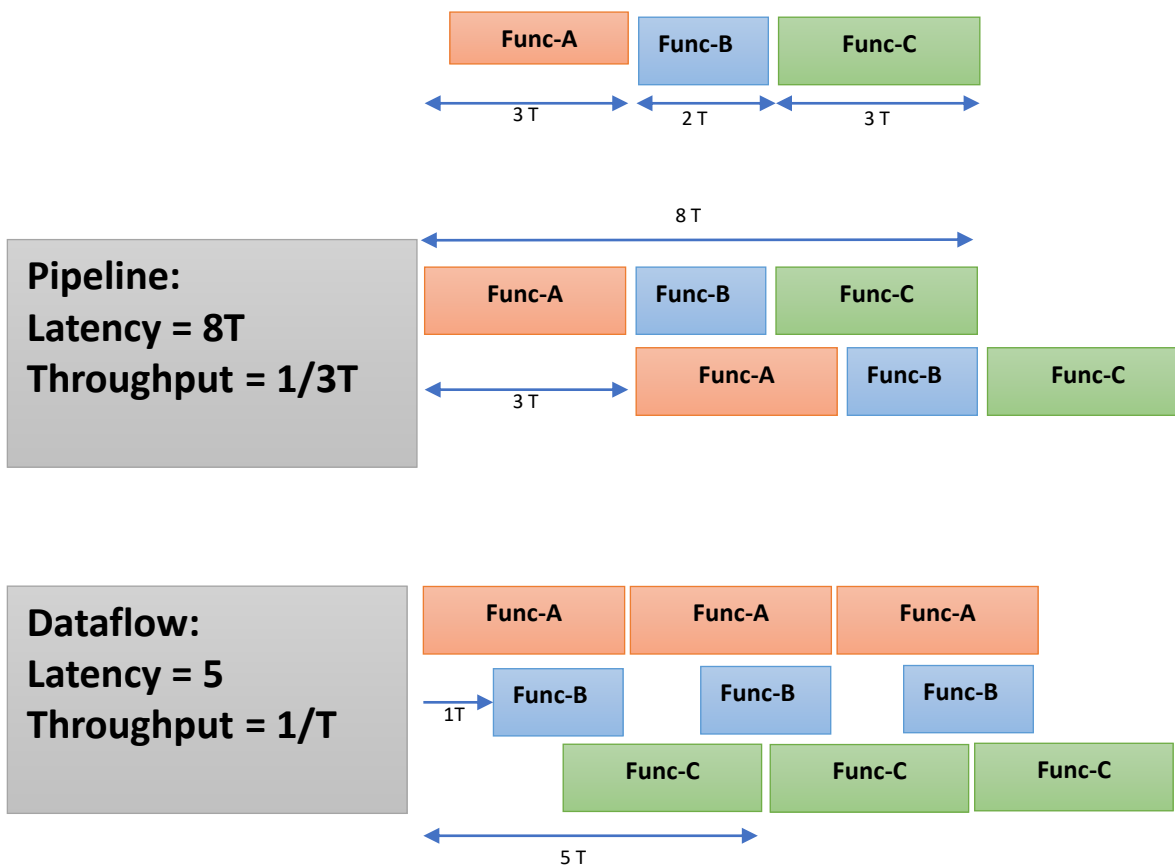
Basic Concept: Latency, Initiation Interval (II)

- Latency = time between start and finish of a task
- Throughput = number of tasks finished in a given time
- Throughput = $1/\text{Latency}$?
- Initiation Interval (II) = Number of clock between new input samples
- Iteration Latency = # of clock to execute one iteration (L)
- Loop Latency = # of clock to execute all the operations in a loop
- Ultimate goal is to achieve $II = 1$ (most critical performance metric)

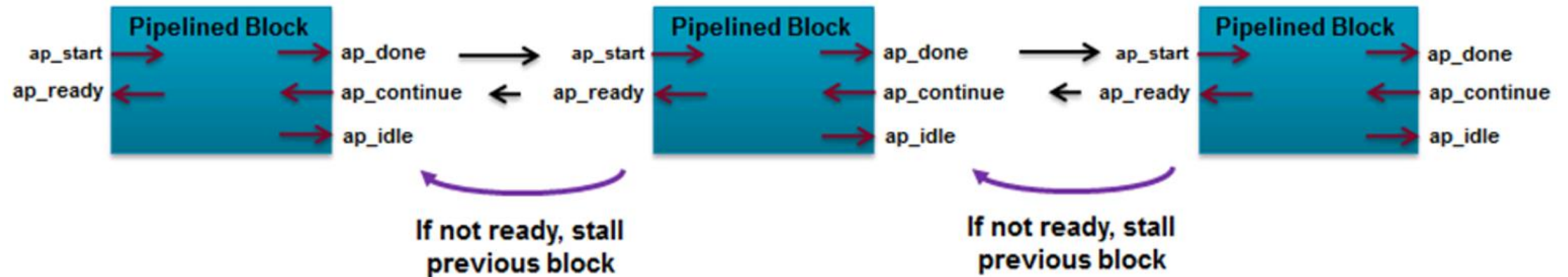
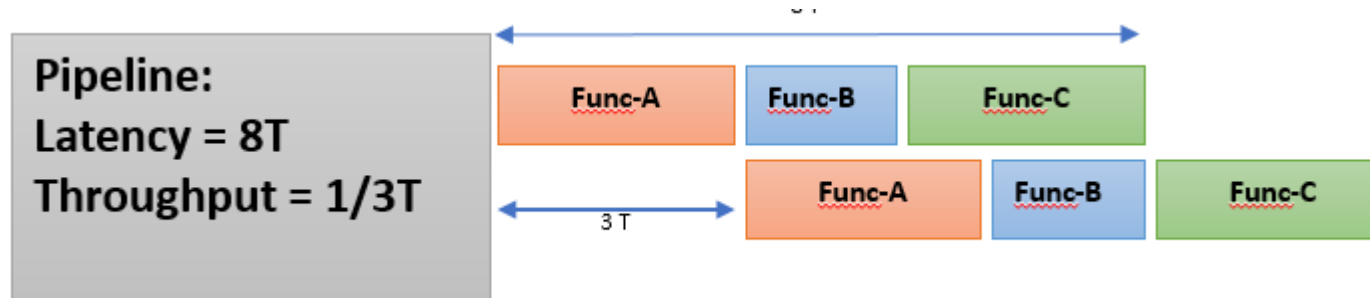


Improve Throughput

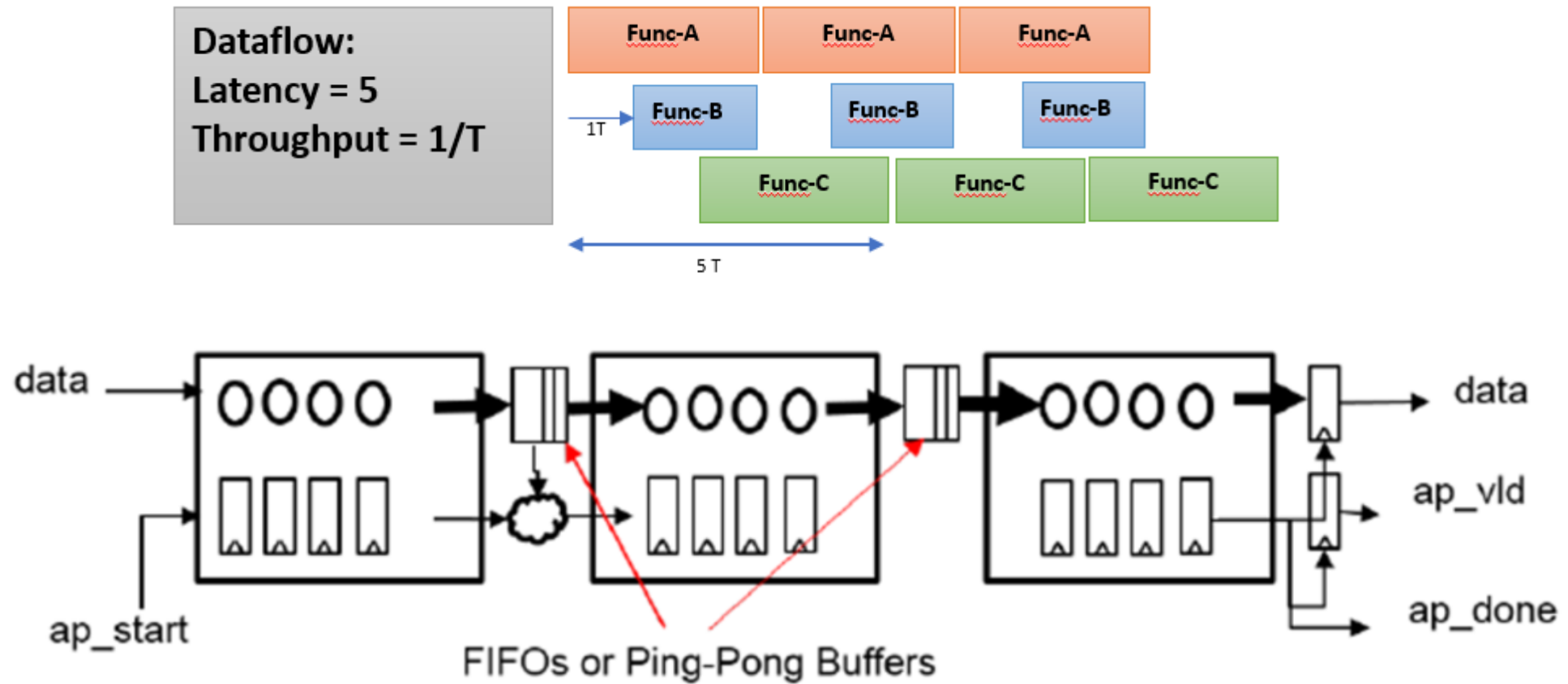
- Data Parallel Processing (SIMD)
 - Replicate hardware, e.g. loop unroll
- Task Level Parallelism
 - Pipeline & Dataflow
 - Decompose function into smaller stages, and allocate separate hardware for each stage
 - Throughput rate depends on II



Pipeline – Kernel IO



Data Flow – Kernel IO



Levels of Pipeline

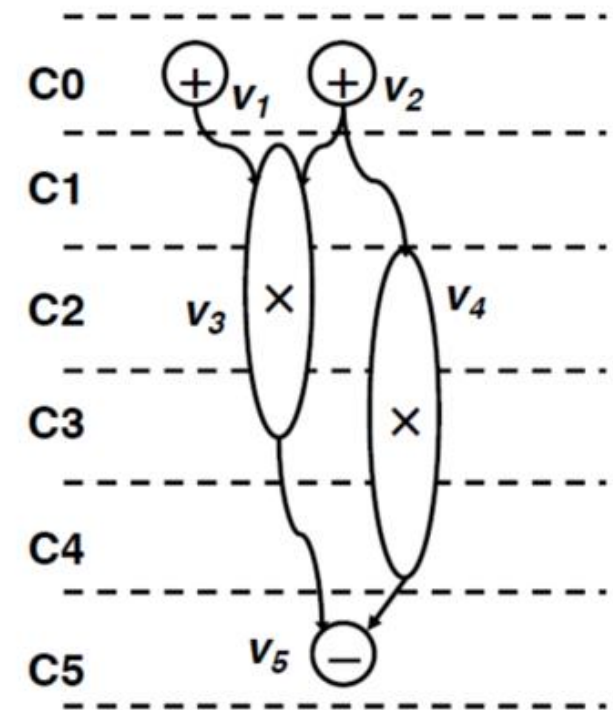
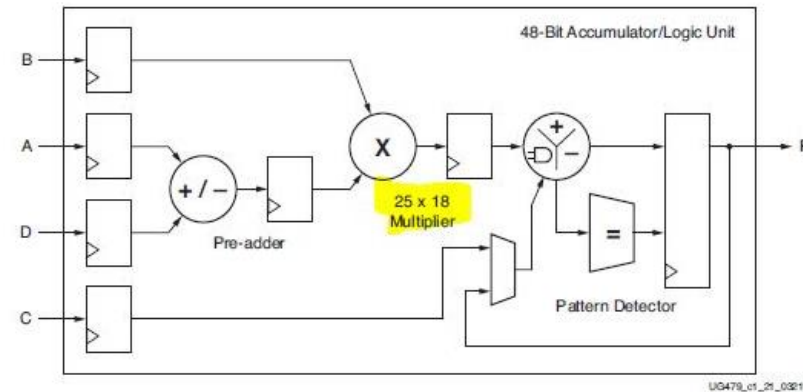
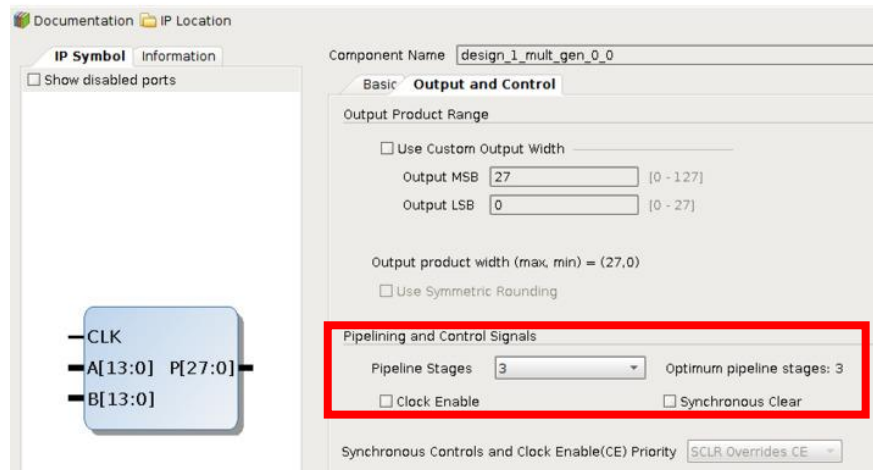
- Operator/Expression pipelining
 - Pipelined function unit (adder, multiplier, memory)
 - Sequence of operations
- Loop pipelining
 - Overlap successive loop iterations
- Function pipelining
 - Overlap function invocations
- Task pipelining
 - Multiple concurrent processes, e.g. multi-thread
 - Start a new task before the prior one completed

Operator/Expression Pipelining

- Pipelined multi-cycle operations
- Xilinx multiplier IP can set # of pipeline stages

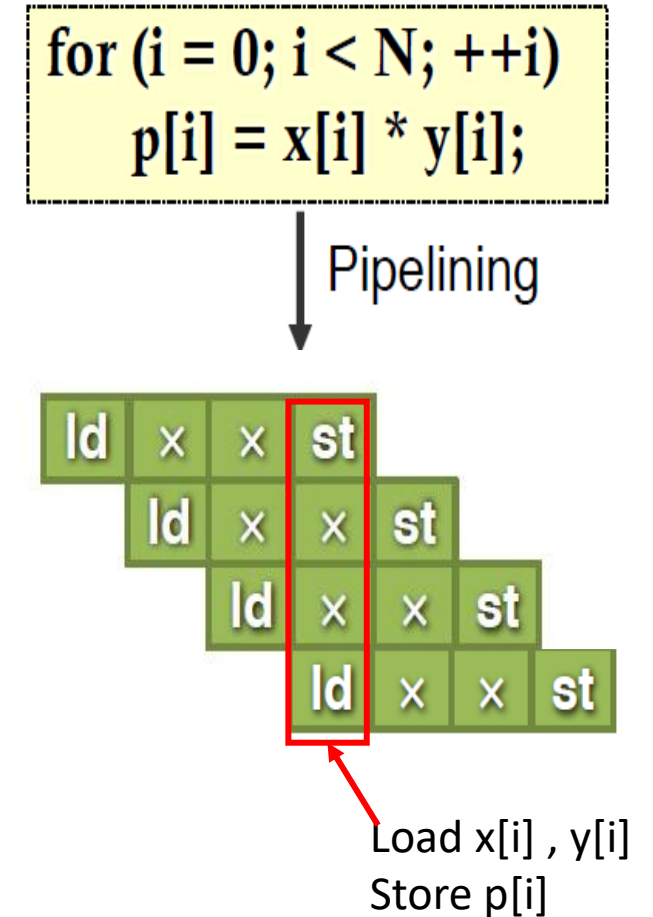
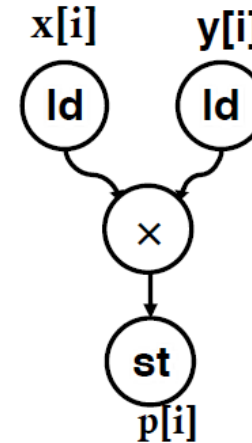
```
v1 = a + b;  
v2 = c + d;  
v3 = v1 * v2;  
v4 = e * v2;  
v5 = v3 - v4;
```

Multiplier (12.0)



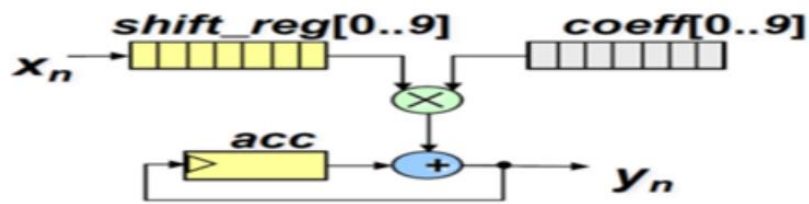
Loop Pipelining

- Start next iteration before current iteration finishes
- Initiation Interval (II) – Key Allow a new iteration to begin every II cycles
- Throughput = $1 / II$
- Resource & data dependency



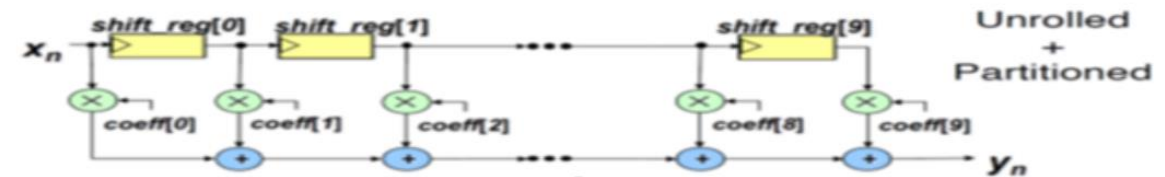
Function Pipelining

```
void fir(data_t *y, data_t x) {
    coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91,
0, 52};
    static data_t shift_reg[N];
    acc_t acc; int i; acc = 0;
    Shift_Accum_Loop: for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x; }
        else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i]; } }
    *y = acc; }
```



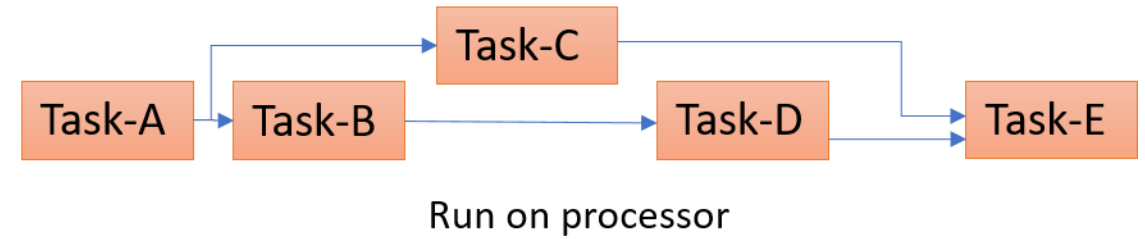
Non-pipelined

```
void fir(data_t *y, data_t x) {
    coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 52}
    #pragma HLS ARRAY_PARTITION variable=c complete dim=1
    static data_t shift_reg[N];
    #pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=1
    acc_t acc; int i; acc = 0;
    TDL:for (i = N - 1; i > 0; i--) {
        #pragma UNROLL
        shift_reg[i] = shift_reg[i - 1];}
    shift_reg[0] = x; acc = 0;
    MAC:for (i = N - 1; i >= 0; i--) {
        #pragma UNROLL
        acc += shift_reg[i] * c[i]; }
    *y = acc;}
```

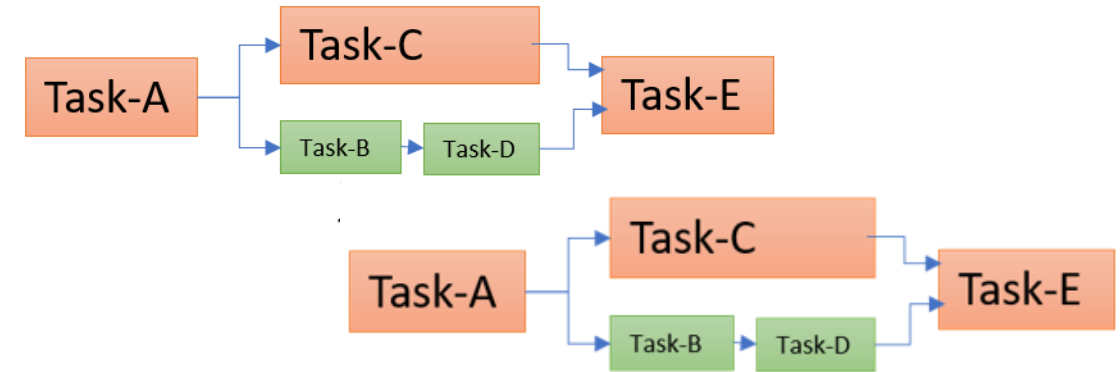


Pipelined

Task Pipelining



- Workflow composed of multiple tasks (Task Dependency Graph)
- Tasks can run at CPU or FPGA
- Task A, C, E run at host CPU, task B, D run at FPGA

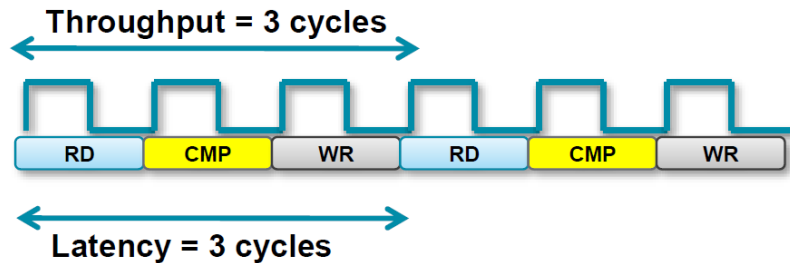


Loop and Function Pipeline

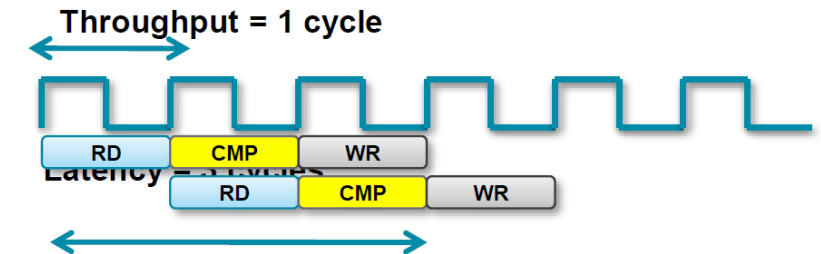
Function Pipelining

- > There are 3 clock cycles before operation RD can occur again
 - >> Throughput = 3 cycles
- > There are 3 cycles before the 1st output is written
 - >> Latency = 3 cycles
- > The latency is the same
 - >> Less cycles, higher throughput

Without Pipelining

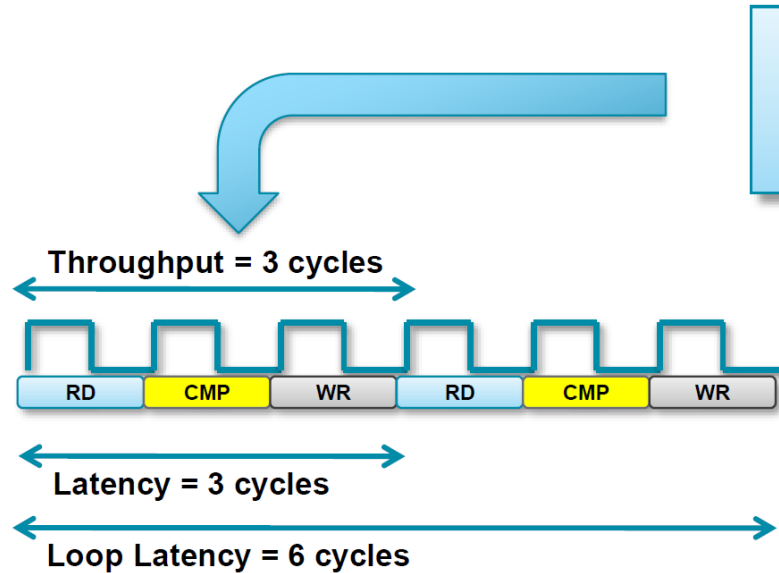


With Pipelining



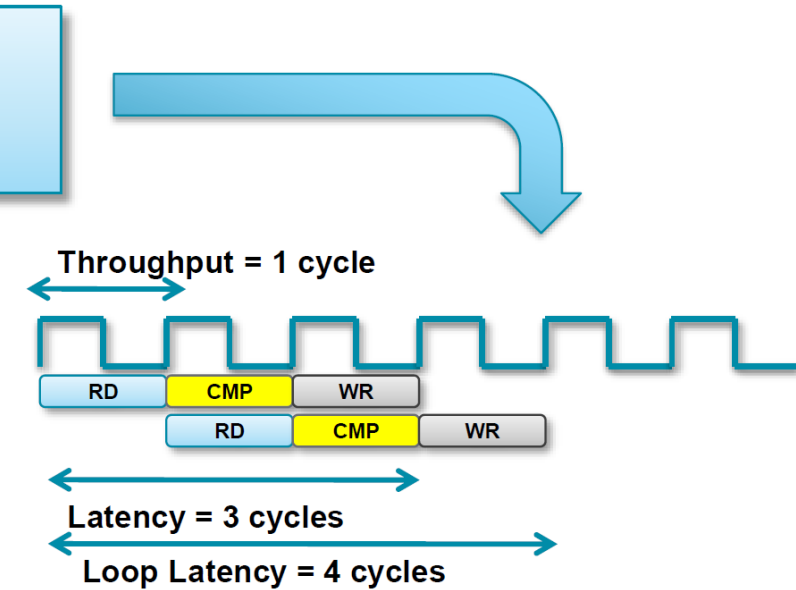
Loop Pipelining

Without Pipelining



- > There are 3 clock cycles before operation RD can occur again
 - >> Throughput = 3 cycles
- > There are 3 cycles before the 1st output is written
 - >> Latency = 3 cycles
 - >> For the loop, 6 cycles

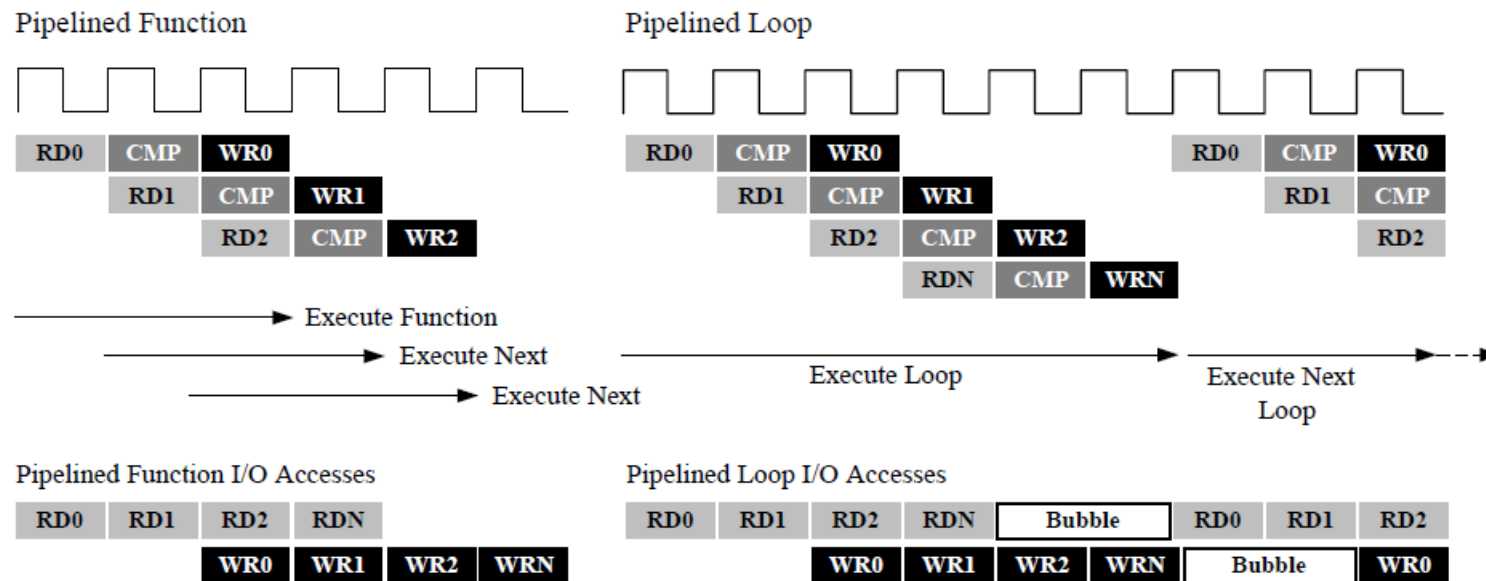
With Pipelining



- > The latency is the same
 - >> The throughput is better
 - Less cycles, higher throughput
- > The latency for all iterations, the loop latency, has been improved

Difference of Function and Loop Pipelining

- Pipeline Function
 - Runs forever and never ends - the functions are automatically rewind.
 - It could unroll all loops in function body
 - Use dataflow parallelism instead
- Pipeline Loop
 - Loop complete all operations in the Loop before starting the next Loop
 - Loop takes overhead (cycles) to enter the Loop, flush the pipeline before entering the next pipeline



X14302

Function Execute Concurrently v.s. Loop Execute Sequentially.

```
void simple_test(const float arr_in[8], float arr_out[8]) {  
#pragma HLS ARRAY_PARTITION variable=arr_out complete dim=1  
#pragma HLS ARRAY_PARTITION variable=arr_in complete dim=1  
  
    outer: for (int i = 0; i < 8; i += 1) {  
#pragma HLS UNROLL  
        float b = sqrt(arr_in[i]); ;  
        inner: for (int j = 0; j < 4; j++) {  
            b += sqrt(b) ;  
        }  
        arr_out[i] = b ;  
    }  
}
```

```
float f1(const float a)  
{  
    float b = sqrt(a); ;  
    inner: for (int j = 0; j < 4; j++) {  
        b += sqrt(b) ;  
    }  
    return(b);  
}  
  
void simple_test(const float arr_in[8], float arr_out[8]) {  
#pragma HLS ARRAY_PARTITION variable=arr_out complete dim=1  
#pragma HLS ARRAY_PARTITION variable=arr_in complete dim=1  
    outer: for (int i = 0; i < 8; i += 1) {  
#pragma HLS UNROLL  
        arr_out[i]=f1(arr_in[i]) ;  
    }  
}
```

Functions run concurrently, unless

- Data dependency
- Resource limitation, e.g. memory port, operators

Pipelining and Function/Loop Hierarchy

> Vivado HLS will attempt to unroll all loops nested below a PIPELINE directive

- >> May not succeed for various reason and/or may lead to unacceptable area
 - Loops with variable bounds cannot be unrolled
 - Unrolling Multi-level loop nests may create a lot of hardware
- >> Pipelining the inner-most loop will result in best performance for area
 - Or next one (or two) out if inner-most is modest and fixed
 - e.g. Convolution algorithm
 - Outer loops will keep the inner pipeline fed

```
void foo(in1[ ][ ], in2[ ][ ], ...) {  
    ...  
    L1:for(i=1;i<N;i++) {  
        L2:for(j=0;j<M;j++) {  
            #pragma HLS PIPELINE  
            out[i][j] = in1[i][j] + in2[i][j];  
        }  
    }  
}
```

1adder, 3 accesses

```
void foo(in1[ ][ ], in2[ ][ ], ...) {  
    ...  
    L1:for(i=1;i<N;i++) {  
        #pragma HLS PIPELINE  
        L2:for(j=0;j<M;j++) {  
            out[i][j] = in1[i][j] + in2[i][j];  
        }  
    }  
}
```

Unrolls L2
M adders, 3M accesses

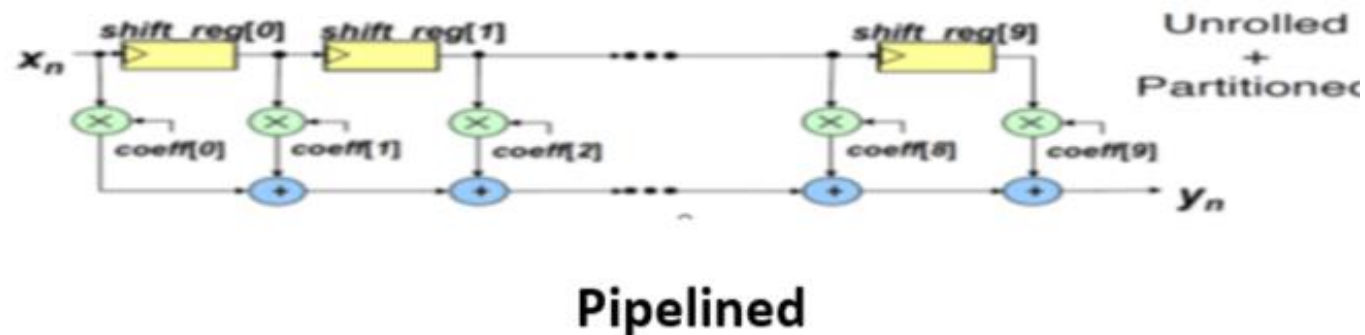
```
void foo(in1[ ][ ], in2[ ][ ], ...) {  
    #pragma HLS PIPELINE  
    ...  
    L1:for(i=1;i<N;i++) {  
        L2:for(j=0;j<M;j++) {  
            out[i][j] = in1[i][j] + in2[i][j];  
        }  
    }  
}
```

Unrolls L1 and L2
N*M adders, 3(N*M) accesses

- Pipeline automatically unroll inner loop
- Pipeline innermost loop gives smallest hardware with generally acceptable throughput

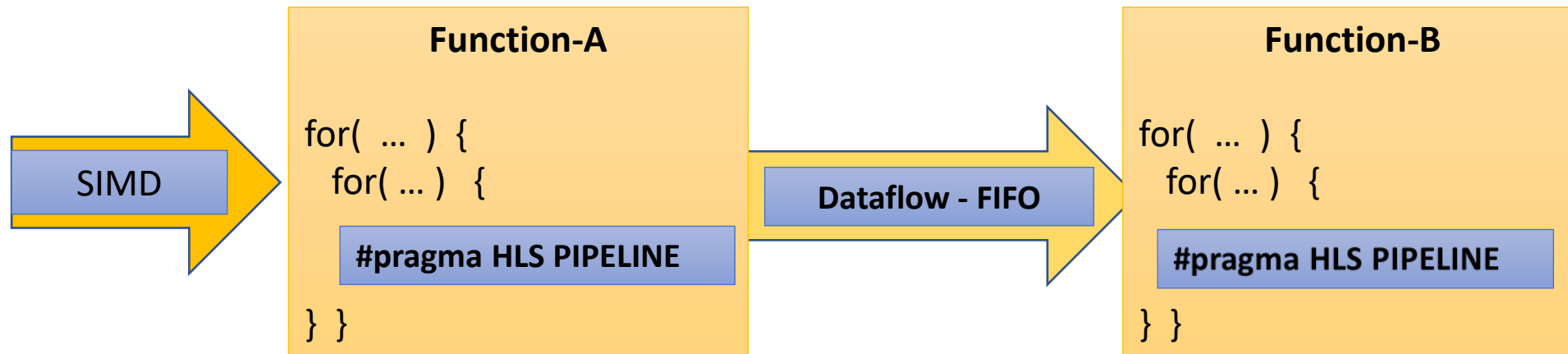
Where we can put the #pragma PIPELINE ?

```
void fir(data_t *y, data_t x) {  
    coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 52}  
    #pragma HLS ARRAY_PARTITION variable=c complete dim=1  
    static data_t shift_reg[N];  
    #pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=1  
    acc_t acc; int i; acc = 0;  
    TDL:for (i = N - 1; i > 0; i--) {  
        #pragma UNROLL  
        shift_reg[i] = shift_reg[i - 1];}  
    shift_reg[0] = x; acc = 0;  
    MAC:for (i = N - 1; i >= 0; i--) {  
        #pragma UNROLL  
        acc += shift_reg[i] * c[i]; }  
    *y = acc;}  
}
```



Best Practice

- Task level parallelism - Function, Loop
 - Pipeline, Dataflow
- Data level parallelism



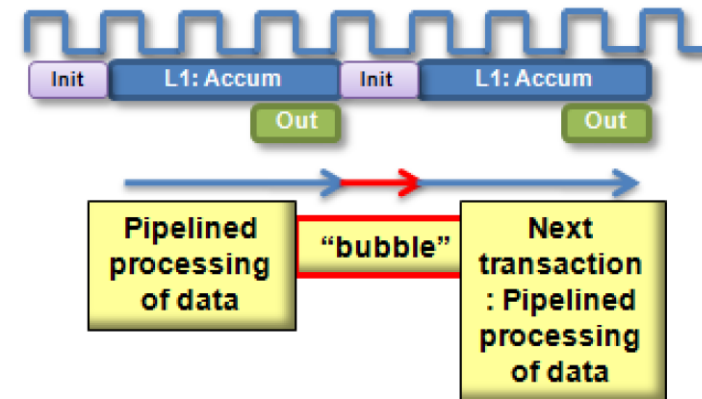
Loop Rewind

Pipelining the Top-Level Loop

> Loop Pipelining top-level loop may give a “bubble”

- >> A “bubble” here is an interruption to the data stream
- >> Given the following

```
void foo_top (in1, in2, ...) {  
    static accum=0;  
    ...  
    L1:for(i=1;i<N;i++) {  
        accum = accum + in1 + in2;  
    }  
    out1_data = accum;  
    ...  
}
```



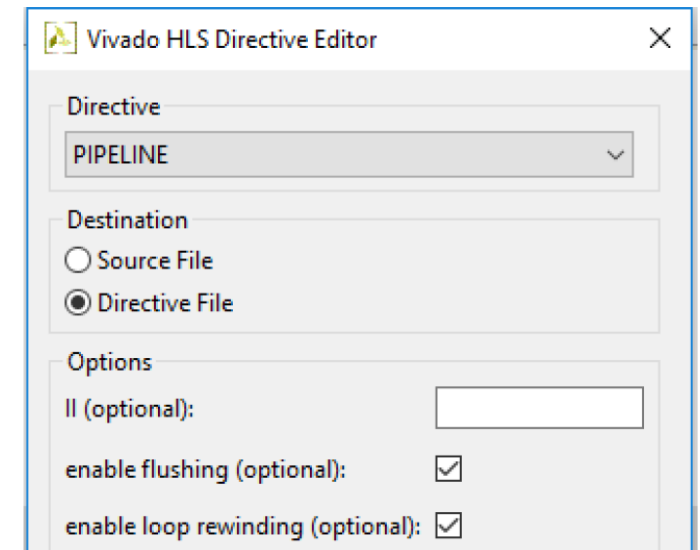
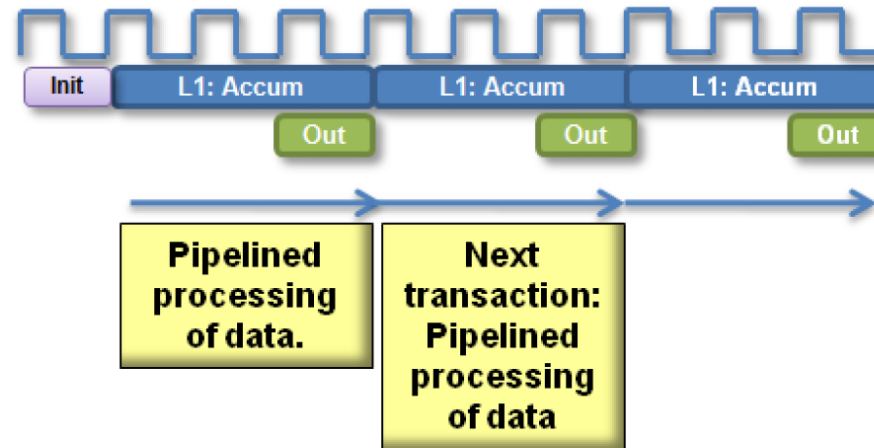
- >> The function will process a stream of data
- >> The next time the function is called, it still needs to execute the initial (init) operations
 - These operations are any which occur before the loop starts
 - These operations may include interface start/stop/done signals
- >> This can result in an unexpected interruption of the data stream

Continuous Pipelining the Top-Level loop

> Use the “rewind” option for continuous pipelining

- >> Immediate re-execution of the top-level loop
- >> The operation rewinds to the start of the loop
 - Ignores any initialization statements before the start of the loop

```
void foo_top (in1, in2, ...) {  
    static accum=0;  
    ...  
    L1:for(i=1;i<N;i++) {  
        accum = accum + in1 + in2;  
    }  
    out1_data = accum;  
    ...  
}
```



> The rewind portion only effects top-level loops

- >> Ensures the operations before the loop are never re-executed when the function is re-executed

Loop Rewinding

Bubble happens

- At loop starts – initialization cycle
- At loop ends – finish all operation before new loop starts

Rewind

- **Only for top-level pipelined loops**
- Pushing any initialization statement before the start of the loop
- These statement cannot contain if-else branches

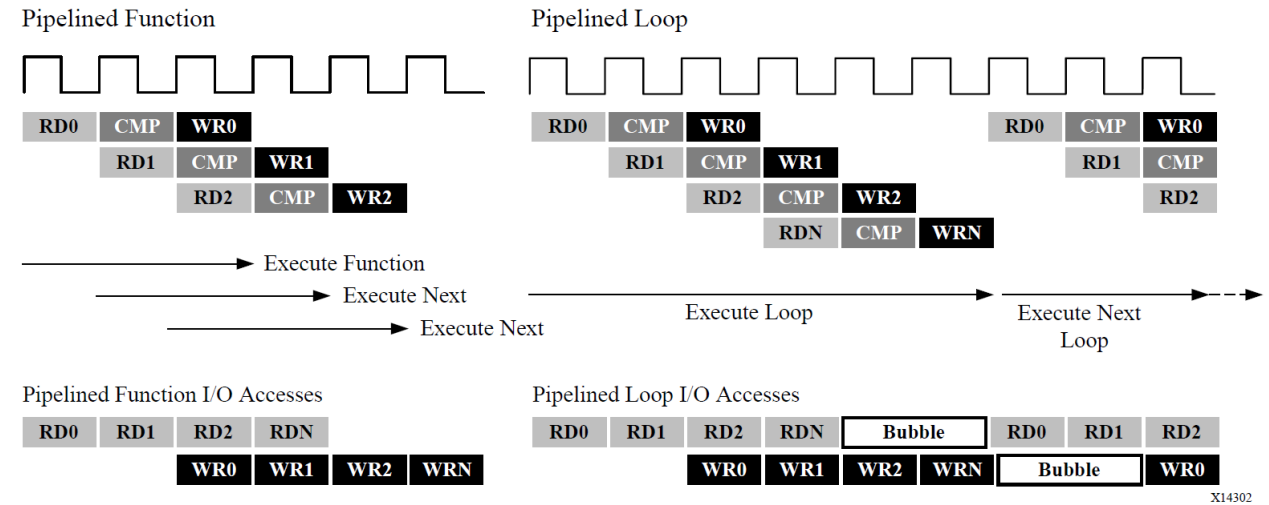
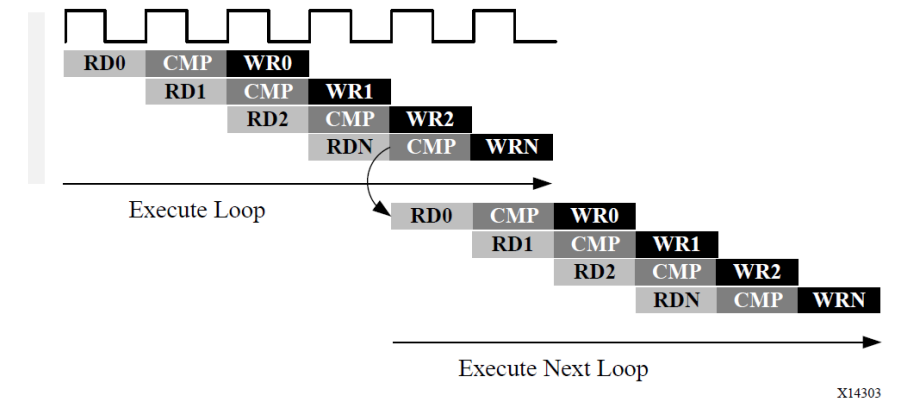


Figure 55: Loop Pipelining with Rewind Option



X14302

X14303

Cases for Rewind Fails

- No code from the start of the function and the begin of the loop
- Allow only single return
- Does not support any continue statement
- At most a single loop
- Nested loops should be flattened

No code from the start of the function and the begin of the loop

```
void decoderInput(hls::stream<mag_type> &inputStream, hls::stream<dataPair> &pairedData)
{
    static count_type cnt;
    mag_type val, val_next;
    dataPair tempData;
    int n=0;
    if(!inputStream.empty()) { // move this code inside the loop
    decoderInput_whileloop: while(1)
    {
        #pragma HLS PIPELINE II=1 rewind
        if(!inputStream.empty()) // move this code inside the loop
        {
            ...
            ...
            ...
        }
    }
}
```

Allow only single return

```
Top (...)  
...  
decoderInput_whileloop:while(1)  
{  
#pragma HLS PIPELINE II=1 rewind  
if(!inputStream.empty()) {  
    if(cnt==0) {  
        tempData.even = inputStream.read();  
        cnt =1;  
    } else {  
        tempData.odd=inputStream.read();  
        pairedData.write(tempData);  
        cnt=0;  
        return 0; // multiple exit condition  
    }  
    } else {  
        return 1; // multiple exit  
    }  
}  
...  
}
```



```
return_value = 1;  
if(condition) {  
    ...  
    return_value = 0;  
}  
return return_value;
```

Single Loop inside the function

- Move loops into each separate function to perform loop rewind
- Flatten the loop

```
decoderInput_whileloop: while(1)
{
    #pragma HLS PIPELINE II=1 rewind
    if(!inputStream.empty())
    {
        ...
    }
    else
    {
        ...
        break;
    }
}
for(auto i=0;i<50;i++)
{
    tempData.even = i*2;
    tempData.odd = i*2+1;
    pairedData.write(tempData);
}
```

Conditional Execution – Move the conditional statement either inside the loop or execute this function conditionally

```
func()
{
  if(var)
  {
    for(...)
    {
      #pragma HLS pipeline rewind
      ...
    }
  }
}
```

Loop Flush

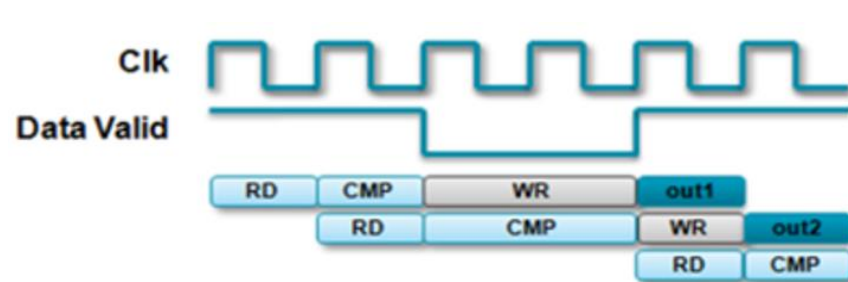
Pipeline Flush

- When there is no data available, the pipeline will stall

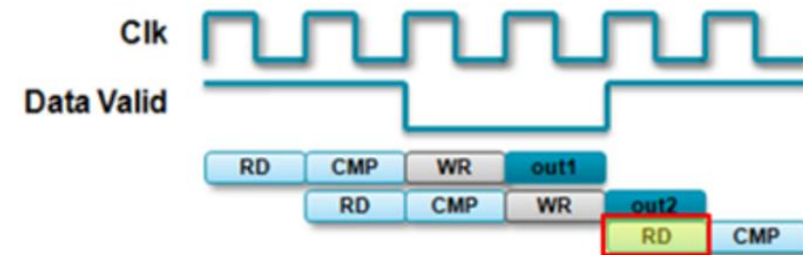
```
#pragma HLS PIPELINE II=<int> enable_flush rewind
```

- Pipeline will flush and empty
- Only for pipelined function**
- Use Flush for Dataflow**

Without Flush: If no data available to proceed (Input Data Valid = 0), the pipeline stalls (the write operation is stalled here)



With flush: When a pipeline is flushed, it continues processing, until reaches to the final stage.



Pipeline Flush Style

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

style = < stp | frp | flp>

- stp: Stall pipeline
- flp: flushable pipeline
 - Consume more resource
 - Can have a larger II because resources cannot be shared among pipeline iterations
- frp : Free-running, flushable pipeline.
 - Run even when input data is not available. May consume more power, pipeline register are clock even if there is no data

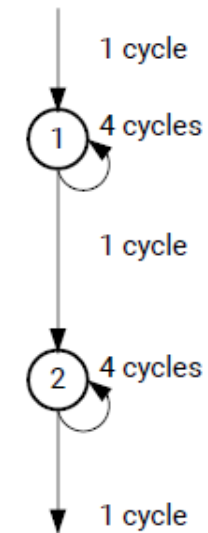
Loop Merge

Loop Latency Calculation

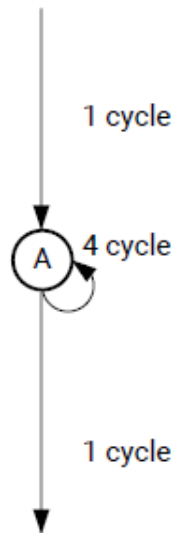
- 1 clock cycle to enter the Add loop.
- 4 clock cycles to execute the Add loop.
- 1 clock cycle to exit Add and enter Sub
- 4 clock cycles to execute the Sub loop
- 1 clock cycle to exit the Sub loop.

```
void top (a[4],b[4],c[4],d[4]...) {  
  
  ... Add: for (i=3;i>=0;i--) {  
    if (d[i])  
      a[i] = b[i] + c[i];  
  }  
  
  Sub: for (i=3;i>=0;i--) {  
    if (!d[i])  
      a[i] = b[i] - c[i];  
  }  
  ...  
}
```

(A) Without Loop Merging



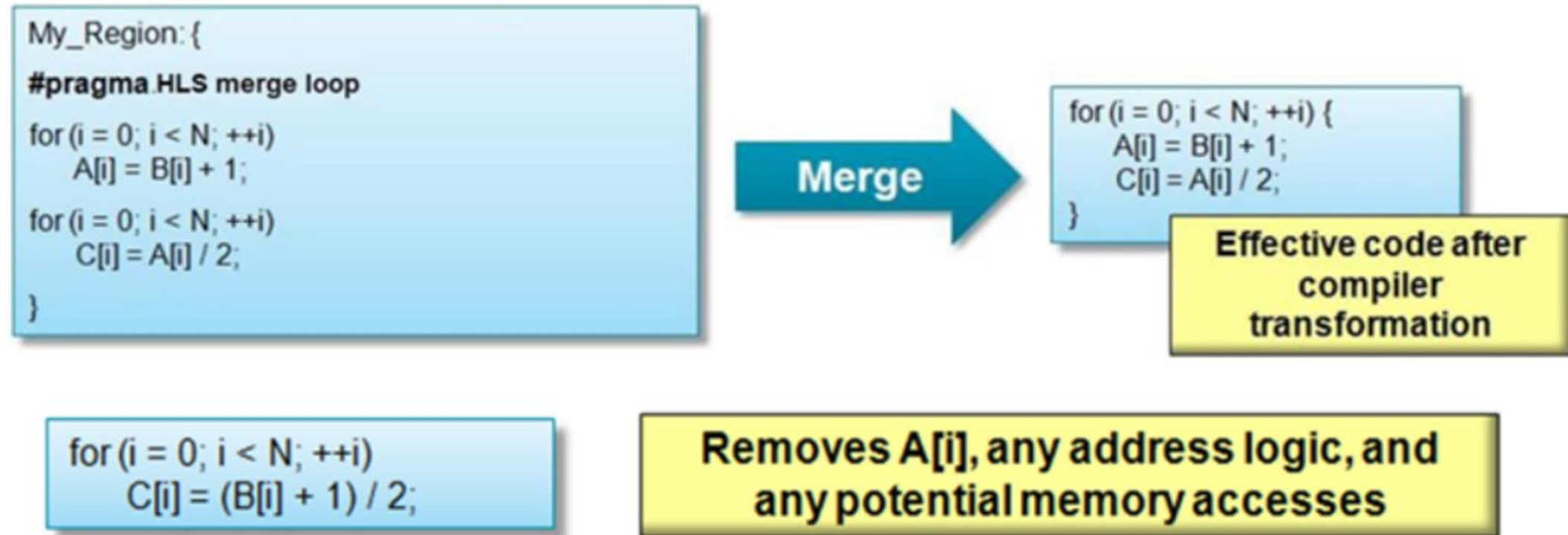
(B) With Loop Merging



X14276-100620

Loop Merge

- Merge sequences of Loops
- Reduce # of clocks transition between loop-body
- Allow logic within loops to be optimized together and implemented parallel



Loop Merge Rules

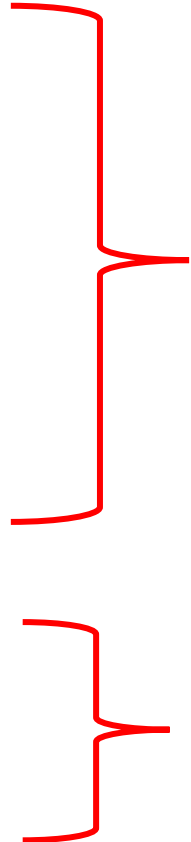
1. If loop bounds are all variables, they must have the same value
2. If loop bounds are constant, the maximum constant value is used as the bound of the merged loop
3. Loops with both variable bound and constant bound cannot be merged
4. Code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results ($a=b$ is allowed; $a=a+1$ is not)
5. Loops cannot be merged when they contain FIFO accesses. Merging would change the order of the reads and writes from a FIFO

Code between loops to be merged cannot have side effects

```
void top (a[4],b[4],c[4],d[4], var, ot...) {  
    ...  
    Add: for (i=3;i>=0;i--) {  
        if (d[i])  
            a[i] = b[i] + c[i];  
    }  
  
    // non-trivial code.  
    int h = var*a[0]*ot;  
  
    Sub: for (i=3;i>=0;i--) {  
        if (!d[i])  
            a[i] = b[i] - c[i] + h ;  
    }  
    ...  
}
```

Loops should not have multiple exit conditions

```
void example_label(int A[50], int B[50]) {  
    int i=0;  
    #pragma HLS loop_merge force  
    do{  
        B[i] = A[i] + 5;  
        if(B[i]==A[i]*B[i]) {  
            break;  
        }  
        else if (B[i] == A[i]+B[i]) {  
            break;  
        }  
        i++;  
    } while(i<50);  
    i=0;  
    do{  
        B[i] = A[i] + 5;  
        i++;  
    } while(i<50);  
}
```

The image shows two red curly braces on the right side of the code. The first brace groups the 'if' and 'else if' statements within the first 'do' loop, indicating multiple exit conditions. The second brace groups the single 'while' loop at the bottom, indicating a single exit condition.

Loop Flatten

Loop Flattening

- #pragma HLS loop_flatten [off]
- **It costs a clock cycle to move between loops in the loop hierarchy**
- Reducing # of cycles for loop operation
- Flattens from the loop with #pragma HLS loop_flatten and all (perfect or semi-perfect) loops above into a single loop
- **Apply LOOP_FLATTEN to the loop body of the inner-most loop**

```
Loop_I: for(i=0;i<20;i++) {  
  Loop_J: for(j=0;j<20;j++) {  
    #pragma HLS loop_flatten  
    operation(i,j);  
  }  
}
```

After flatten →

```
Loop_I: for(k=0;k<400;k++) {  
  i = k / 20;  
  j = k % 20;  
  operation(i, j);  
}
```

Loop Type

```
Loop_outer: for (i=3;i>=0;i--) {  
    Loop_inner: for (j=3;j>=0;j--) {  
        [loop body]  
    }  
}
```



Perfect Loop

- Only innermost loop has loop body
- No logic between the loop
- All loop bounds constant

```
Loop_outer: for (i=3;i>N;i--) {  
    Loop_inner: for (j=3;j>=0;j--) {  
        [loop body]  
    }  
}
```

Semi-perfect Loop

- Outermost loop bound a variable

```
Loop_outer: for (i=3;i>N;i--) {  
    [loop body]   
    Loop_inner: for (j=3;j>=M;j--) {  
        [loop body]   
    }  
}
```

Imperfect Loop

- Loop body is not exclusively inside the innermost loop
- The inner loops has variable bounds


Can be flattened

Turn imperfect loop into perfect loop

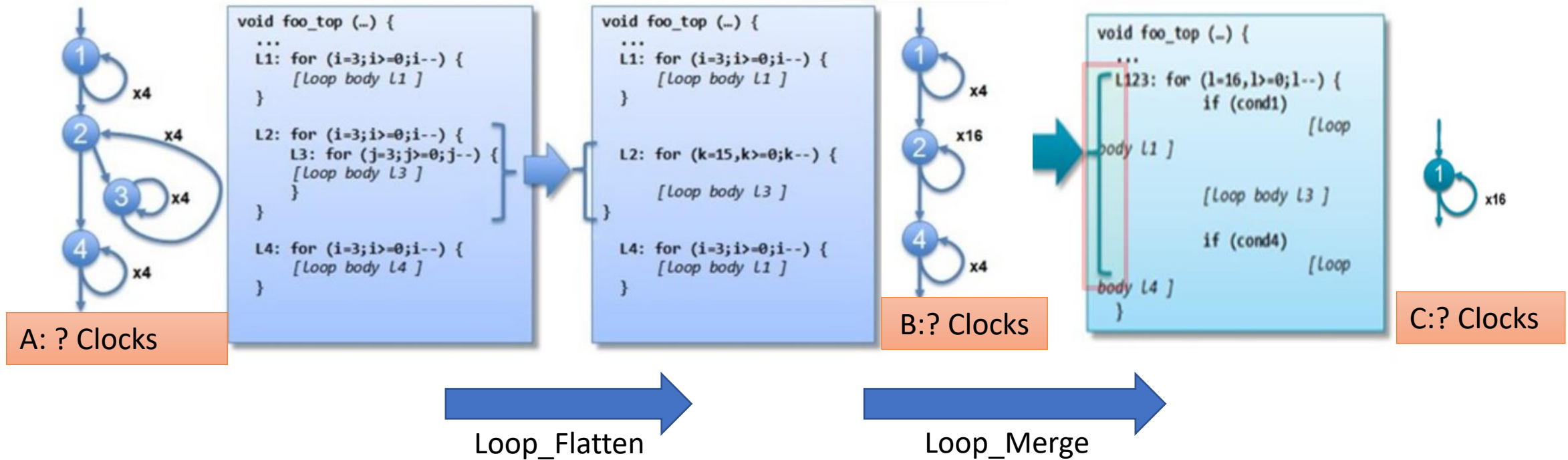
```
void loop_imperfect(din_t A[N], dout_t B[N]) {
    int i,j;
    dint_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        acc = 0; // loop body not completely in inner loop
        LOOP_J: for(j=0; j < 20; j++){
            acc += A[j] * j;
        }
        if (i%2 == 0) // loop body not completely in inner loop
            B[i] = acc / 20;
        else
            B[i] = 0;
    }
}
```

```
void loop_perfect(din_t A[N], dout_t B[N]) {
    int i,j;
    dint_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        LOOP_J: for(j=0; j < 20; j++){
            if(j==0) acc = 0;
            acc += A[j] * j;
            if(j==19) {
                if (i%2 == 0)
                    B[i] = acc / 20;
                else
                    B[i] = 0;
            }
        }
    }
}
```

Question#1 – Calculate # of Cycle from Flatten & Merge

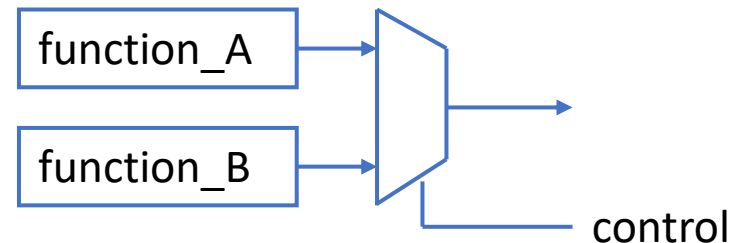


PIPELINE Limitation

Issues to Prevent Pipeline, Loop

- Loops with Variable Bounds
- Data Hazard: Data Dependency in inter/intra-iteration - Recurrence
- Structure Hazard: Resource Limitation (Contention)
 - Limited compute resources
 - Limited I/O bandwidth
 - Limited Memory resources, e.g. port limitation
- Note: Control Hazard (Branch) is not an issue in FPGA

```
if( control )  
    function_A( ... );  
else  
    function_B( ... );
```



Loops with Variable Bounds

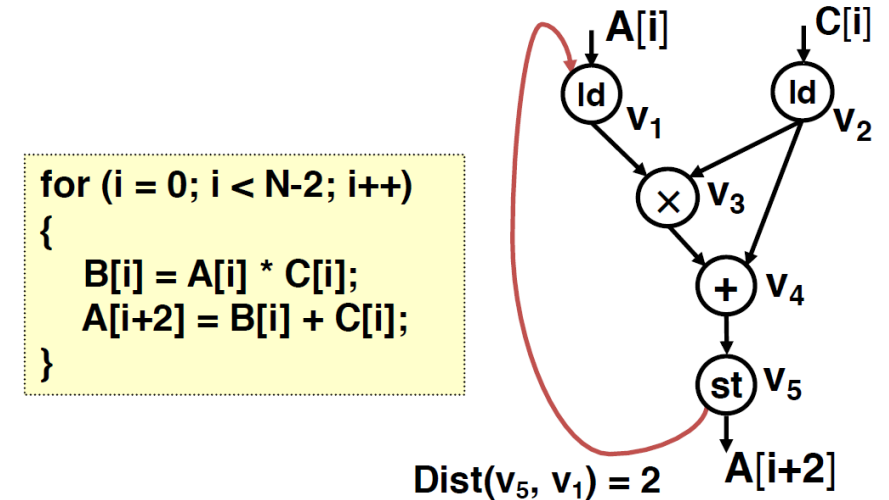
- Loops with variable bounds cannot be unrolled, because pipeline automatically unroll the loop.
- Re-code the design to remove variable bounds or use the max bounds with an exit statement
- Can not report loop latency
 - Use Tripcount directive
 - `#pragma HLS loop_tripcount nin=<int> max=<int>`
 - Use assert macro in C- code
 - `asserts(variable < N)`
 - Only for reporting purpose

```
int loop_variable_bounds(int A[N], int width) {  
  
    LOOP_X: for( x=0; x<width; x++) {  
        out = += A[x];  
    }  
}
```

```
#define N 32  
int loop_max_bounds(int A[N], int width) {  
  
    assert(width < N);  
    LOOP_X: for (x=0; x<N; x++) {  
        if( x< width) {  
            out = += A[x]  
        }}  
    }  
}
```

Calculate II

- Dependency Graph
- Latency(c): sum of operation latencies along c
- Distance(c) = iterations separating the two dependent operations
- Ops(r) = # of operation for operator r
- Resource(r) = # of operator available
- ResMII – Minimum II due to Resource Limits (**Structure Hazard**)
 - $\text{ResMII} = \max [\text{Ops}(r)/\text{Resource}(r)]$
 - r = operator
- RecMII – Minimum II due to Recurrences (**Data Hazard**)
 - **$\text{RecMII} = \max [\text{Latency}(c)/\text{Distance}(c)]$**
 - **Optimize by reduce Latency(c), or increase Distance(c)**
- $\text{Min-II} = \max(\text{ResMII}, \text{RecMII})$



		Latency ->				
Iteration->	I/A0	*	*	+	s/A2	
	I/A1		*	*	+	s/A3
	I/A2			*	*	+

Types of Data Hazard

- RAW (Read After Write)
- Write-after-Read (WAR)
- Write-After-Write (WAW)

Note: Only RAW is a true hazard.

Data Dependency – RAW (Read After Write)

- **True dependency**
- S1-iteration(u) -> S2-iteration (v)
- S1 computes a value S2 uses

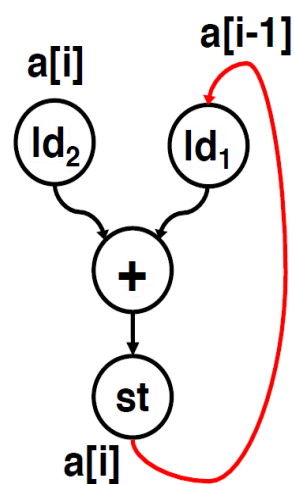
```
for(i=0; i < N; i++)  
{  
    A[i] = A[i-1] * a;  
}
```

II=1							
iteration#	C1	C2	C3	C4	C5	C6	C7
i=1	L/A[0]	*	S/A[1]				
i=2		L/A[1]	*	S/A[2]			
i=3			L/A[2]	*	S/A[3]		
II=3							
i=1	L/A[0]	*	S/A[1]				
i=2				L/A[1]	*	S/A[2]	
i=3							

Recurrence – Prefix Sum

Recurrence:

- Computation depends on a previous result
- $\Pi = ?$

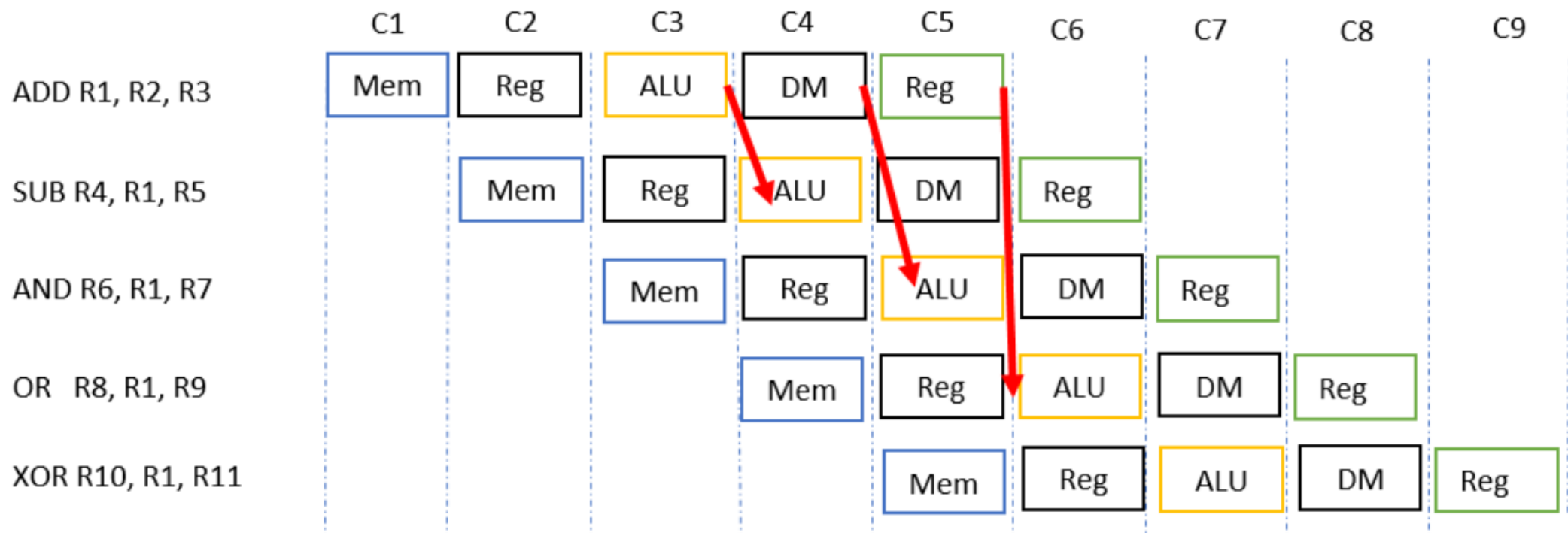


```
for (i = 1; i < N; ++i)
  a[i] = a[i-1] + a[i];
```

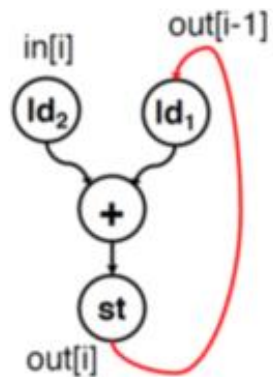
	cycle 1	cycle 2	cycle 3	cycle 4
$i = 0$	ld ₁ ld ₂	+	st	
$i = 1$	$\Pi = 1$	ld ₁ ld ₂	+	st

Data Forwarding

A result passes directly to the functional unit that requires it without going through input/output, i.e., memory access.



Data Forwarding – Prefix-Sum

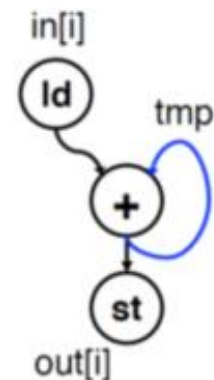


ld – Load
st – Store

```
out[0] = in[0];
for ( int i = 1; i < N; i++ )
    out[i] = out[i-1] + in[i];
```

	cycle 1	cycle 2	cycle 3	cycle 4
$i = 0$	ld ₁ ld ₂	+	st	
$i = 1$	// = 1	ld ₁ ld ₂	+	st

Assume chaining is not possible on memory reads (i.e., ld) and writes (i.e., st) due to cycle time constraint



ld – Load
st – Store

```
int tmp = in[0];
for ( int i = 1; i < N; i++ ) {
    tmp += in[i];
    out[i] = tmp;
}
```

	cycle 1	cycle 2	cycle 3	cycle 4
$i = 0$	ld	+	st	
$i = 1$	// = 1	ld	+	st

Data Dependency – Write-after-Read (WAR)

- **WAR (Write After Read) – anti dependence**
 - S1 read from a memory location update by S2
 - Renaming to resolve WAR

```
For( ... i++) {  
    A[i-1] = b - a;  
    B[i] = A[i] + 1;  
}
```

II=1							
iteration#	C1	C2	C3	C4	C5	C6	C7
i=1	S/A[0]	L/A[1]	+	S/B[1]			
i=2		S/A[1]	L/A[2]	+	S/B[2]		
i=3			S/A[2]	L/A[3]	+	S/B[3]	
II=2							
i=1	S/A[0]	L/A[1]	+	S/B[1]			
i=2			S/A[1]	L/A[2]	+	S/B[2]	
i=3							

Data Dependency – Write-After-Write (WAW)

- WAW (Write-After-Write) – Output dependence
- S1 write to a memory that write by S2
- Renaming to resolve WAW

```
For( ... i++) {  
    B[i] = A[i-1] + 1;  
    A[i] = B[i+1] + b  
    B[i+2] = b - a; }
```

II=1							
iteration#	C1	C2	C3	C4	C5	C6	C7
i=1	L/A[0]	S/B[1]	L/B[2]	S/A[1]	S/B[3]		
i=2		L/A[1]	S/B[2]	L/B[3]	S/A[2]	S/B[4]	
i=3			L/A[2]	S/B[3]	L/B[4]	S/A[3]	S/B[5]



Register Renaming

- register renaming (in hardware)
 - change register names to eliminate WAR/WAW hazards

key: think of architectural registers as names , not locations

- can have more locations than names
- dynamically map names to locations
- **map table** holds the current mappings (name→location)
 - write: allocate new location and record it in map table
 - read: find location of most recent write by name lookup in map table
 - minor detail: must de-allocate locations appropriately

Register Renaming Examples

- names: r1, r2, r3, locations: 11, 12, 13, 14, 15, 16, 17
- original mapping: r1→11, r2→12, r3→13 (14-17 "free")

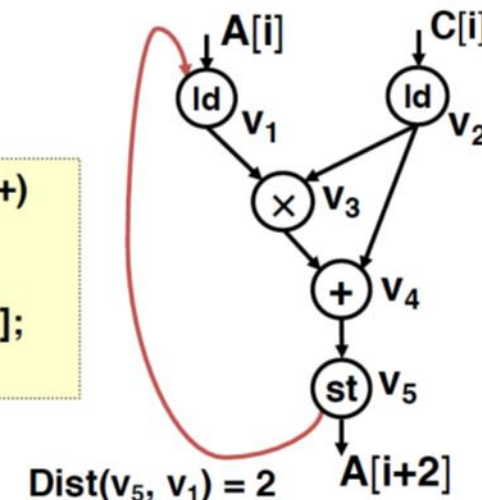
raw instructions	map table	free locations	renamed instructions																		
	<table><tr><th>r1</th><th>r2</th><th>r3</th></tr><tr><td>11</td><td>12</td><td>13</td></tr><tr><td>14</td><td>12</td><td>13</td></tr><tr><td>14</td><td>12</td><td>15</td></tr><tr><td>16</td><td>12</td><td>15</td></tr><tr><td>16</td><td>17</td><td>15</td></tr></table>	r1	r2	r3	11	12	13	14	12	13	14	12	15	16	12	15	16	17	15	14, 15, 16, 17	
r1	r2	r3																			
11	12	13																			
14	12	13																			
14	12	15																			
16	12	15																			
16	17	15																			
add r1, r2, r3		15, 16, 17	add 14, 12, 13																		
sub r3, r2, r1		16, 17	sub 15, 12, 14																		
mul r1, r2, r3		17	mul 16, 12, 15																		
div r2, r1, r3			div 17, 16, 15																		

- renaming removes WAW/WAR, leaves RAW intact!!

Reduce II – Increase the Distance of dependent variables

- $\text{RecMII} = \max [\text{Latency}(c)/\text{Distance}(c)]$
- Apply Tile Interleaving to increase Distance
- The following code

```
for (i = 0; i < N-2; i++)  
{  
    B[i] = A[i] * C[i];  
    A[i+2] = B[i] + C[i];  
}
```



datatype is double, it takes 6T for */+ => II = 6

```
for(int i = j+1; i < diagSize; ++i){  
    dataType tmp2=0;  
    Loop vec mul:  
    for(int k = 0; k < j; k++){  
        tmp2 += dataA[i][k]*dataA[j][k];  
    }  
    dataA[i][j] = (dataA[i][j] - tmp2)/dataA[j][j];  
}
```

Distance = 16, so II = 1

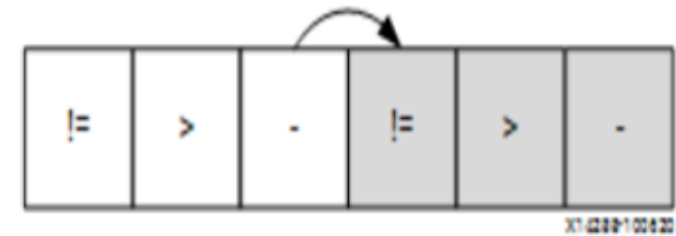
```
for(int i = j+1; i < diagSize; ++i){  
    dataType tmp_i[16] = {0}, tmp3_i, tmp1[8], tmp2[4],  
    Loop_vec_mul:  
    for(int k = 0; k < j; k++){  
        #pragma HLS pipeline  
        tmp_i[k % 16] += dataA[i][k]*dataA[j][k];  
    }  
}
```

Scalar Dependencies

- It can not less than the latency to calculate exit condition
- Loop with data-dependent control flow
- Can we re-structure the code to make $II = 1$?
- Speculative/Dynamic Pipeline

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

Figure: Scalar Dependency



Exit condition takes multiple cycles

```
while ( m*m*m < N ) {  
    m += 1;  
}
```

Remove False Dependency

- Restructure code to remove dependency
- `#pragma HLS DEPENDENCE variable=hist intra RAW false`

```
void histogram(int in[INPUT_SIZE], int
hist[VALUE_SIZE]) {
#pragma HLS DEPENDENCE variable=hist inter
RAW distance=2
    int val;
    int old = -1;
    for(int i = 0; i < INPUT_SIZE; i++) {
#pragma HLS PIPELINE
        val = in[i];
        assert(old != val);
        hist[val] = hist[val] + 1;
        old = val;
    }
}
```

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int acc = 0; int val, old = in[0];
    #pragma HLS DEPENDENCE variable=hist intra RAW false
    for(i = 0; i < INPUT_SIZE; i++) {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val) {
            acc = acc + 1;
        } else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}
```

#pragma HLS dependence variable=<variable> <class> \
<type> <direction> distance=<int> <dependent>

- **variable**=<variable>
- **class** = [array | pointer]
- **type**=[inter | intra]
 - Loop-independent dependence (Intra) : the same elements is accessed in a single loop iteration
 - Loop-carried dependency (Inter) : same element accessed from a different loop iteration.
- **direction** = [RAW | WAR | WAW]
- **distance**=<int>
- **dependent**=[true | false]

```
for (i=0;i<N;i++) {  
    A[i]=x;  
    y=A[i];  
}
```

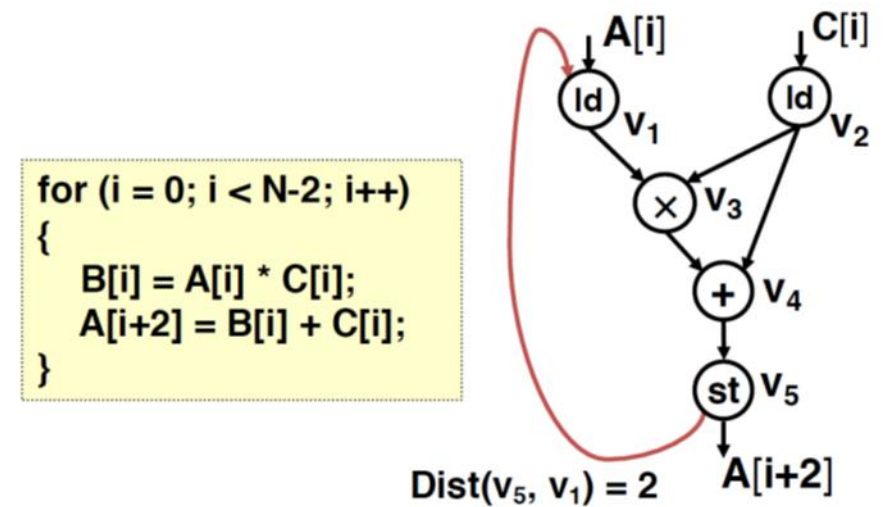
```
for (i=0;i<N;i++) {  
    A[i]=A[i-1]*2;  
}
```

Question

Refer to $\text{RecMII} = \max [\text{Latency}(c)/\text{Distance}(c)]$

Data Forwarding Techniques is to optimize

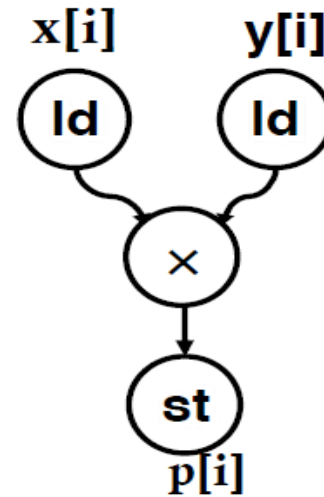
- Term: $\text{Latency}(c)$?
- Term: $\text{Distance}(c)$?



Resource Limitation, Recurrence

Resource Limitation

- Memory port
 - dual-port RAM
 - Array_Partition
- Operator/Function



```
for (i = 0; i < N; ++i)  
  p[i] = x[i] * y[i];
```

Pipelining



Load `x[i]`, `y[i]`
Store `p[i]`

Unroll

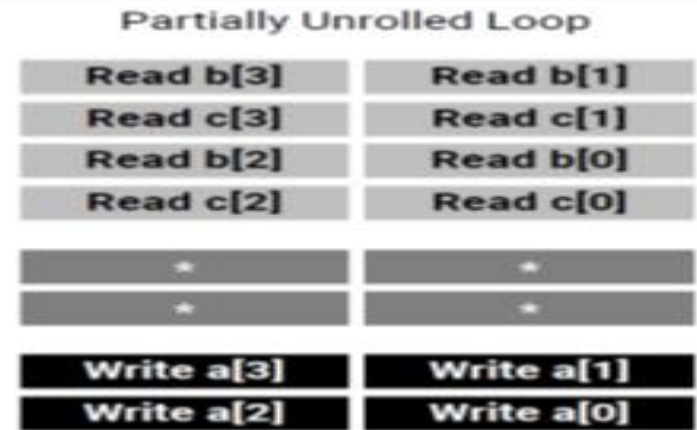
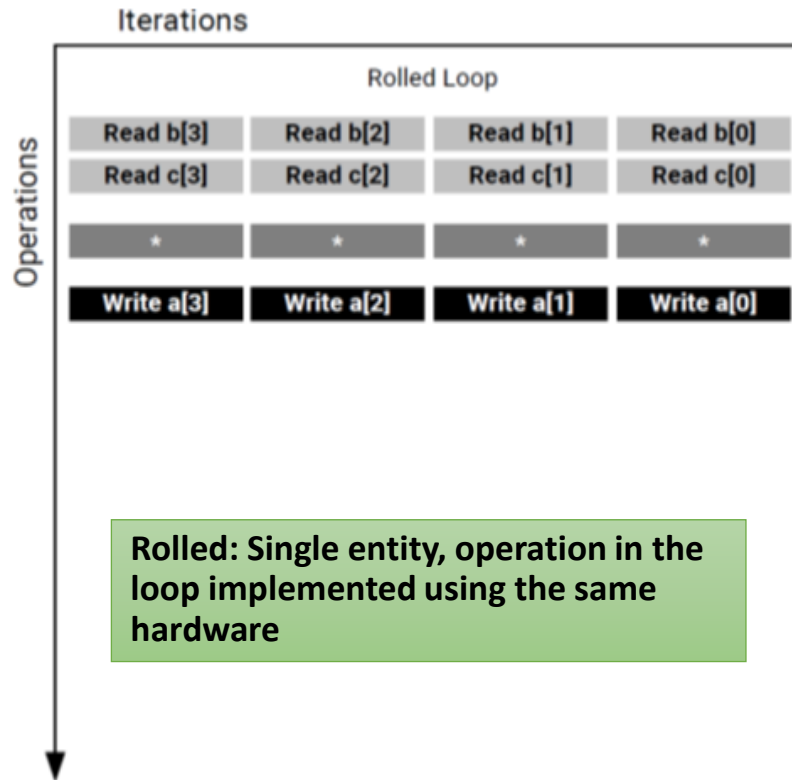
```
#pragma HLS UNROLL factor=<N> region skip_exit_check
```


Illustration of Rolled, Unrolled, Partial unroll

```
for_mult: for (i=3; i>0;i--) {  
    a[i] = b[i] * c[i];  
}
```

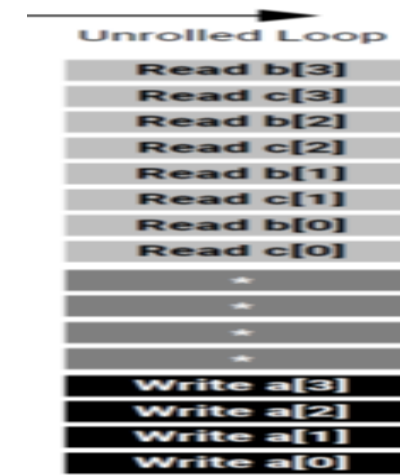
```
for_mult: for (i=3; i>0;i--) {  
    #pragma UNROLL factor = 2  
    a[i] = b[i] * c[i];  
}
```

```
for_mult: for (i=3; i>0;i--) {  
    #pragma UNROLL  
    a[i] = b[i] * c[i];  
}
```



Partially Unrolled: factor = 2

- Data dependency



All the iteration are treated as multiple independent operations

- Data dependency
- Resource

Loop with Variable Bounds

```
// width is from the input
dout_t loop_variable_bounds(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;
}
```

Partial unrolling with skip_exit_check

```
Loop_2: for(i = 0; i<N;i++) {  
#pragma HLS unroll factor=2  
    a[i] = b[i] + c[i];  
}
```



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

```
Loop_2: for(i = 0; i<N;i++) {  
#pragma HLS unroll skip_exit_check factor=2  
    a[i] = b[i] + c[i];  
}
```



```
for(i= 0; i < N; i +=2) {  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1]  
}
```

Illustration of Roll with region

Unroll all loops within the body (region) of the specified loop, without unrolling the enclosing loop itself. (unrolls loop2, loop_3, but not loop_1)

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
    int temp1[N];  
    loop_1: for(int i = 0; i < N; i++) {  
  
        #pragma HLS unroll region  
        temp1[i] = data_in[i] * scale;  
        loop_2: for(int j = 0; j < N; j++) {  
            data_out1[j] = temp1[j] * 123;  
        }  
        loop_3: for(int k = 0; k < N; k++) {  
            data_out2[k] = temp1[k] * 456;  
        }  
    }  
}
```

Question#2: Which pipeline design will you choose?

A

```
Int loop_pipeline(int A[N]) {  
    Loop_I: for(i=0; i < 20; i++) {  
        Loop_J: for(j=0; j < 20; j++) {  
            #pragma HLS pipeline  
            acc += A[i] * j;  
        }  
    }  
}
```

B

```
Int loop_pipeline(int A[N]) {  
    Loop_I: for(i=0; i < 20; i++) {  
        #pragma HLS pipeline  
        Loop_J: for(j=0; j < 20; j++) {  
            acc += A[i] * j;  
        }  
    }  
}
```

C

```
Int loop_pipeline(int A[N]) {  
    #pragma HLS pipeline  
    Loop_I: for(i=0; i < 20; i++) {  
        Loop_J: for(j=0; j < 20; j++) {  
            acc += A[i] * j;  
        }  
    }  
}
```