



Design • Optimize • Verify

Loop Optimizations

Catapult Basic Training Module 4

Mike Fingeroff & Rich Toone

Agenda

- Loops and C++ style
- Loop Optimizations
 - Unrolling
 - Pipelining
 - Merging
- Loops and Memories
- Pipeline Feedback Analysis

Note: See Chapter 4 of the High-level Synthesis Blue Book

Loops in High Level Synthesis

- Loops are the primary mechanism for applying high level synthesis constraints as well as moving data, or I/O, into and out of an algorithm
 - “for” / “while” / “do ... while”
- One of the most important features of HLS for tuning design performance is Loop Unrolling and Loop Pipelining
- In order to talk about how to write loops it's helpful to introduce a few definitions:
 - **Loop iterations** - the number of times the loop runs before it exits
 - **Loop iterator** - the variable used to compute the loop iteration
 - **Loop body** - the code between the start and the end of the loop
 - **Loop unrolling** – effectively copying the body of the loop with the loop iterator replaced by its value for each iteration
 - **Loop pipelining** – starting the next iteration of a loop after a set time (possibly before the previous iteration has finished)

Loops in High Level Synthesis

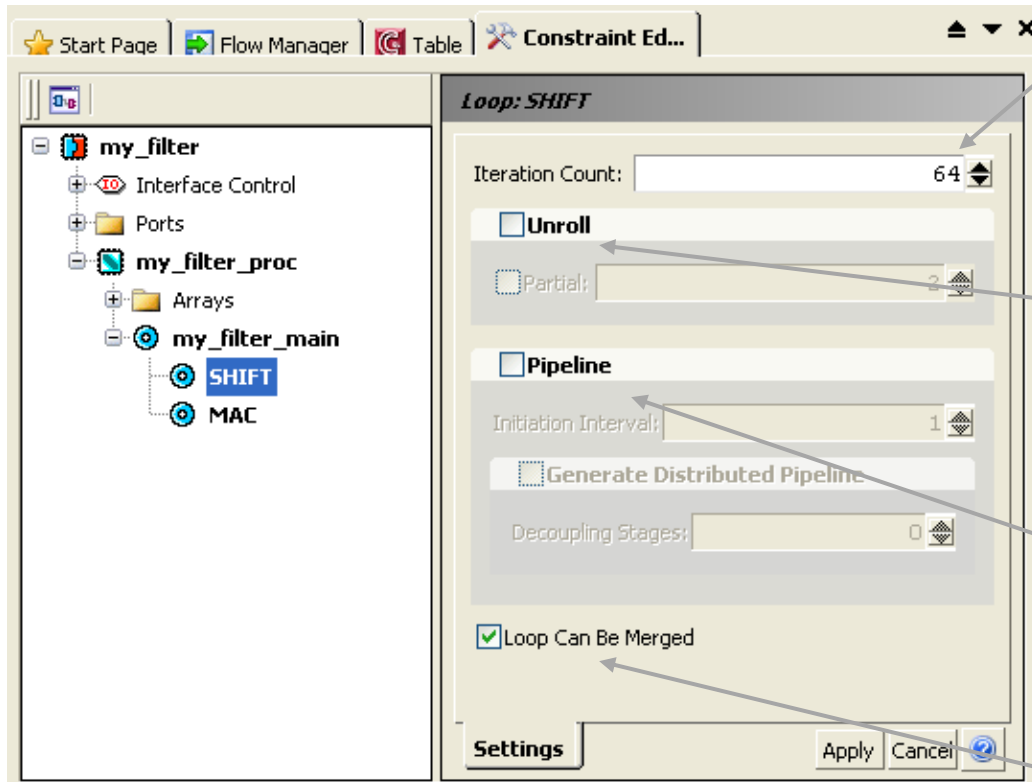
- Every design always has at least one “loop”
 - The function call itself (Also called the “Main” loop)
- Loops can be labeled in your C++ code
 - MAC:for (int i=0;i<16;i++)
- Loops are a central area for optimization
 - Iterations – can be specified for unknown limits
 - Unrolling – the number of times the loop body is copied (with implied concurrency)
 - Pipelining – how often to start the next iteration
 - Merging – allowed to operate in parallel or sequentially
- A Loop synthesized by Catapult comprises an iterator/counter controlling a scheduled data path
 - Each level of loops involves a counter
- The style in which loops are written can have a significant impact on the quality of results of the generated hardware

What Defines a Loop?

- Any loop (“for”, “while”, “do ... while”) consists of four parts:
 - Initialization** - which consists of zero or more comma-delimited variable initialization statements
 - Test-condition** - which is evaluated to determine if the execution of the for loop continues
 - Increment** - which consists of zero or more comma-delimited statements that increment variables
 - Statement-list** - which consists of zero or more statements that execute each time the loop is executed.
- Example (“for” loop):

```
LABEL: for( initialization; test-condition; increment ) {  
    statement-list;  
}
```

Loop Architectural Constraints



- Iteration count
 - Is it correct?
 - Is it known?
- Unroll
 - Partial unroll
- Pipeline
 - Initiation Interval
- Loop Merging

Agenda

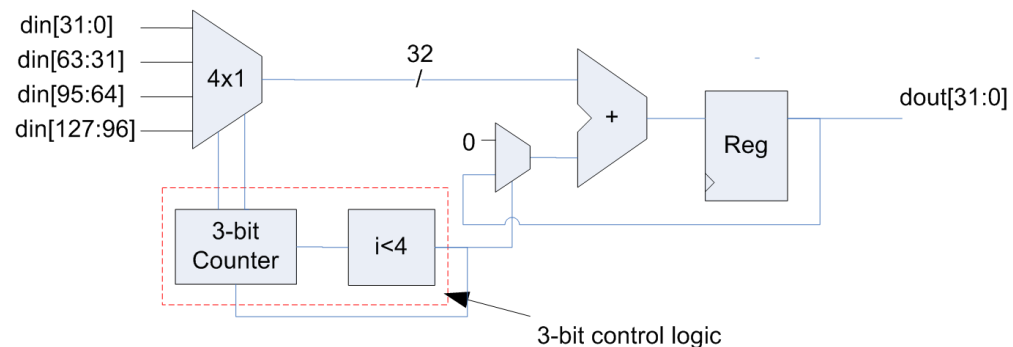
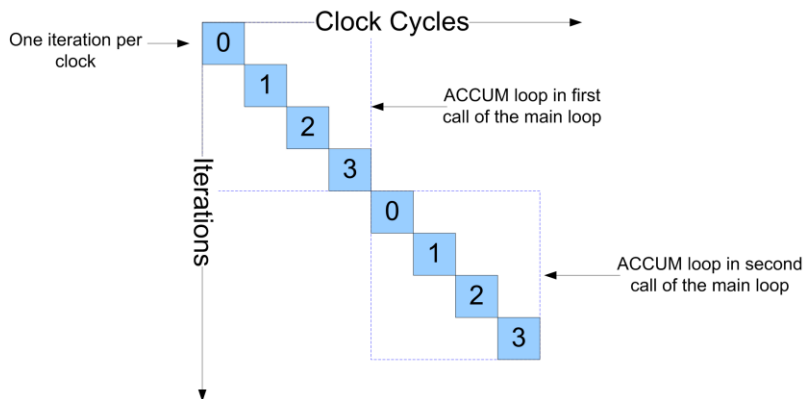
- Loops and C++ style
- Loop Optimizations
 - Unrolling
 - Pipelining
 - Merging
- Loops and Memories
- Pipeline Feedback Analysis

Rolled Loops

- Unconstrained loops are considered to be “rolled”
 - If a loop is left “rolled”, each iteration of the loop takes at least one clock cycle to execute in hardware
 - There is an implied “wait until clock” for the loop body

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    ACCUM:for(int i=0;i<4;i++){  
        acc += din[i];  
        <Implied wait-until-clock>  
    }  
    dout = acc;  
}
```

Design Constraints:
All loops left rolled



Loop Unrolling

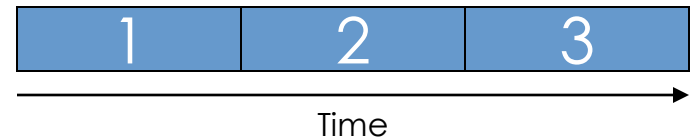
- Loop unrolling is the primary mechanism to add parallelism into a design
 - Done by automatically scheduling multiple loop iterations in parallel, when possible
 - Amount of parallelism is controlled by how many loop iterations are run in parallel
- Loop unrolling can theoretically execute all loop iterations within a single clock cycle, as long as there are no:
 - Dependencies between successive iterations
 - Accesses to restricted bandwidth resources (such as memories)

Loop Unrolling Theory

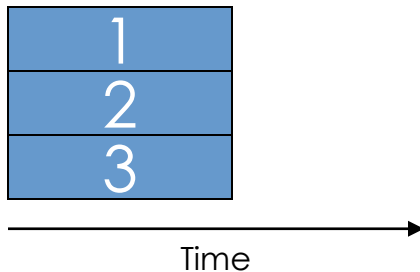
Unrolling a loop copies the loop body:



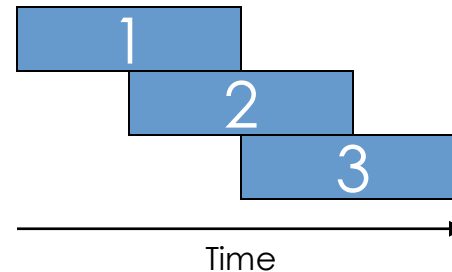
If there are looping dependency chains, unrolling doesn't help:



If there are no dependency chains, all loop bodies can execute in parallel:



Usually the result is somewhere in the middle:



Fully Unrolled Loops

- Fully unrolling the ACCUM loop allows all iterations to be scheduled in the same clock cycle

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    ACCUM:for(int i=0;i<4;i++){  
        acc += din[i];  
    }  
    dout = acc;  
}
```

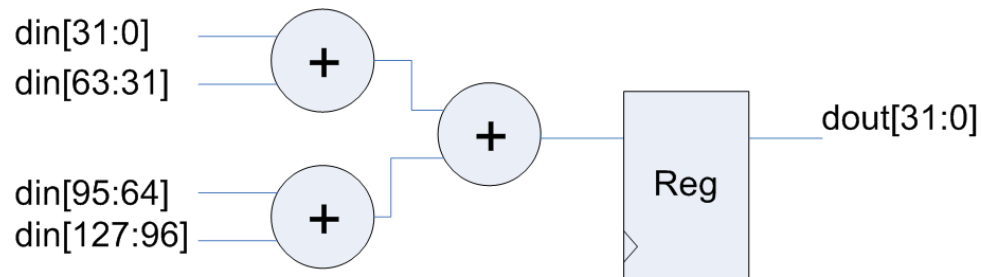
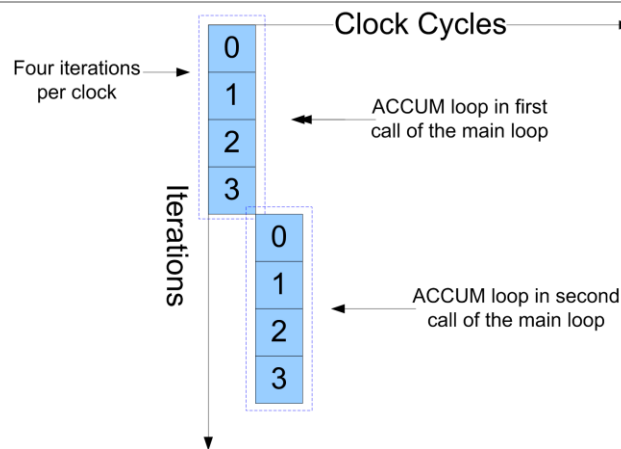
Design Constraints:

Clock freq. slow enough to ignore dependencies

Fully unroll "ACCUM"

Equivalent "manual" unroll:

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    acc += din[0];  
    acc += din[1];  
    acc += din[2];  
    acc += din[3];  
    dout = acc;  
}
```



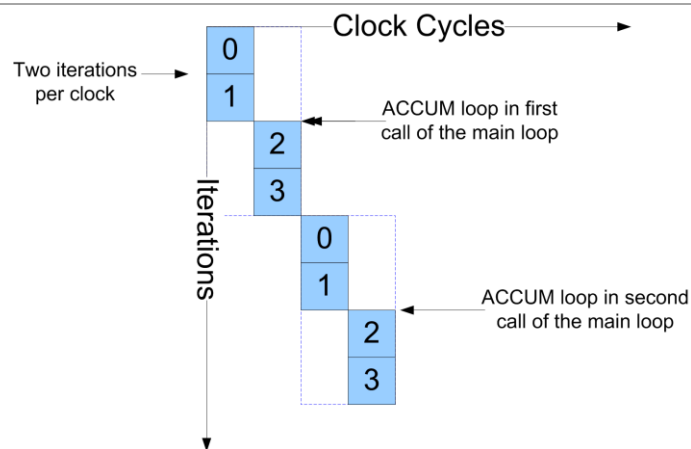
Partial Loop Unrolling

- Unroll by two makes two copies of the loop body and tries to schedule them in parallel

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    ACCUM:for(int i=0;i<4;i++){  
        acc += din[i];  
    }  
    dout = acc;  
}
```

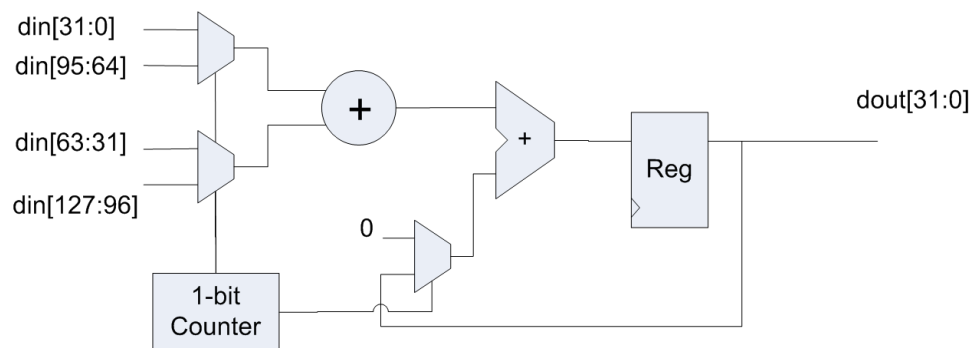
Design Constraints:

Clock freq. slow enough to ignore dependencies
Partially unroll "ACCUM" by two

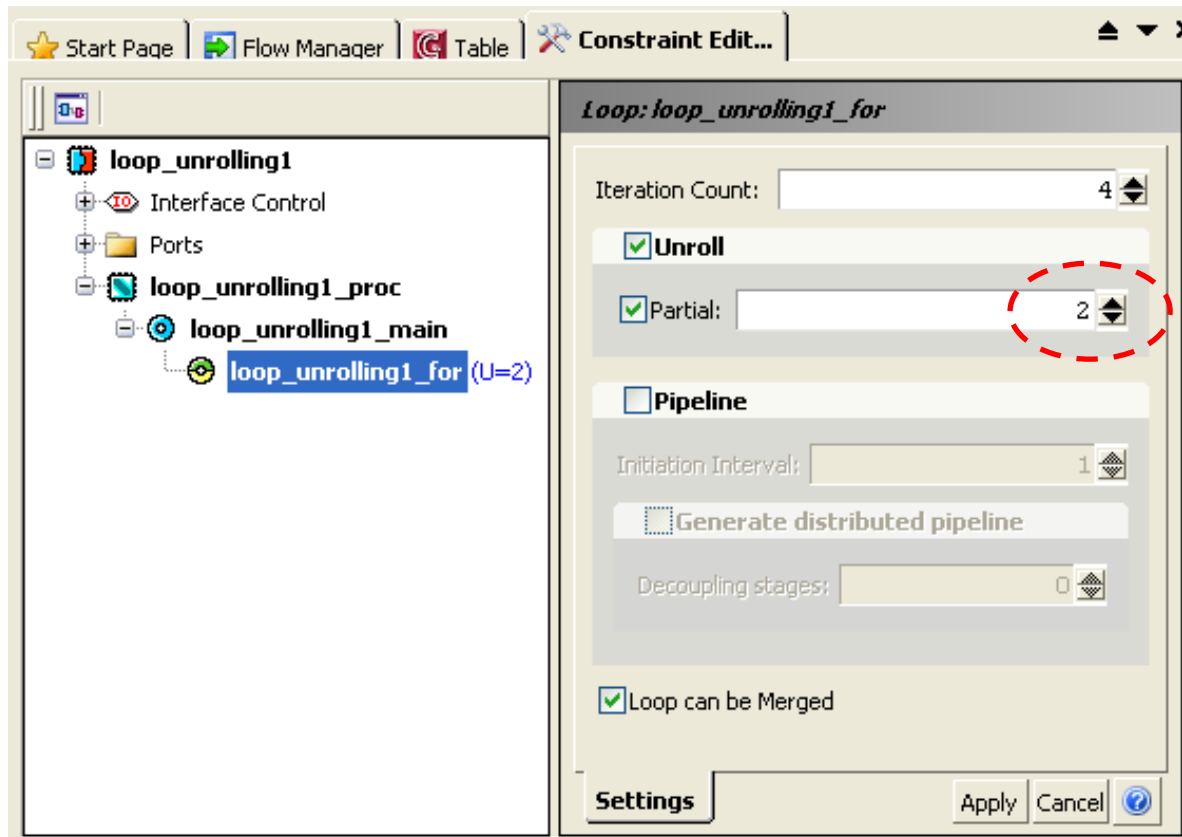


Equivalent "manual" unroll by two:

```
void accumulate(int din[4], int &dout){  
    int acc=0;  
    ACCUM:for(int i=0;i<4;i+=2){  
        acc += din[i];  
        acc += din[i+1];  
    }  
    dout = acc;  
}
```



Setting Loop Unrolling Constraints



- Partial unrolling

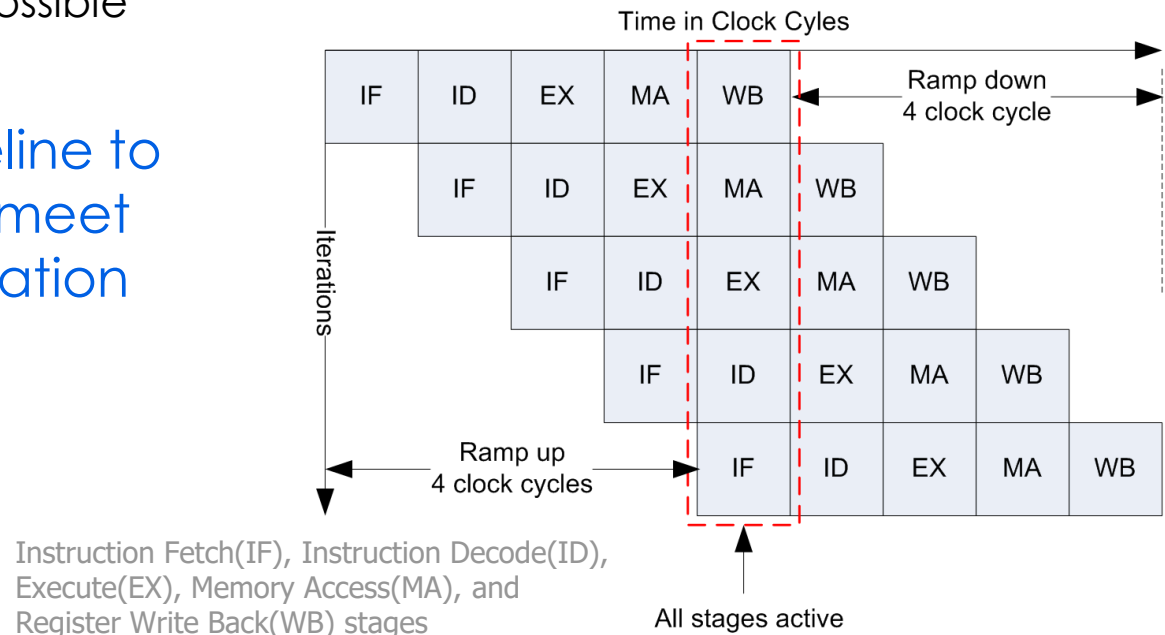
- Implement 'n' copies in parallel
- Powers of 2, or whole divisors typically work well

Loop Unrolling – When & How to Unroll

- Be methodical
 - Don't just unroll everything
- Always start with the innermost loops of nested loops
 - Then work your way out
- Look for loops with the following properties
 - Small number of iterations
 - Small number of operations
 - No big feedback loops
 - Few limited resources (i.e. RAMS)
- Avoid unrolling a loop with rolled sub-loops
 - Creates sequential loops and complex control logic

Loop Pipelining

- "Loop Pipelining" is similar to the classic RISC pipeline covered in most introductory computer architecture classes
 - RISC pipelining allows a new instruction to be fetched each clock cycle while the other pipeline stages are gradually activated
 - Difference between the RISC pipeline and HLS loop pipelining is that the RISC pipeline is designed to fetch and execute every clock cycle
 - A design that does not need to run every clock cycle under-utilizes the pipeline, and a design that needs to fetch and execute multiple times per clock cycle is not possible
- HLS allows the pipeline to be custom built to meet the design specification



Loop Pipelining

- "Loop Pipelining" allows a new iteration of a loop to be started before the current iteration has finished
- Top-level function call has an implied loop, also known as the main loop

Main loop

```
void accumulate(int a, int b, int c, int d, int &dout){  
    int t1,t2;  
    t1    = a  + b;  
    t2    = t1 + c;  
    dout  = t2 + d;  
}
```

- "Loop pipelining" allows the execution of the loop iterations to be overlapped, increasing the design performance by running them in parallel
 - Amount of overlap is specified by the "Initiation Interval (II)"

Loop Pipelining

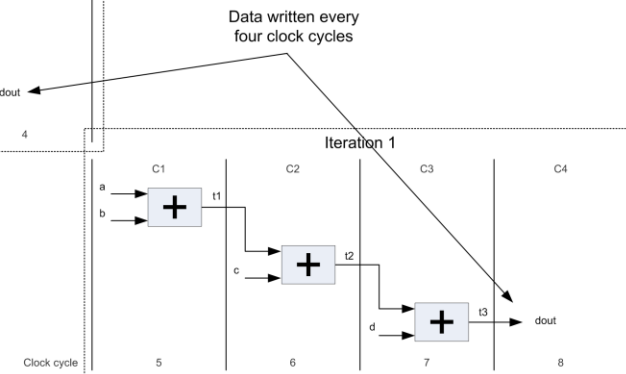
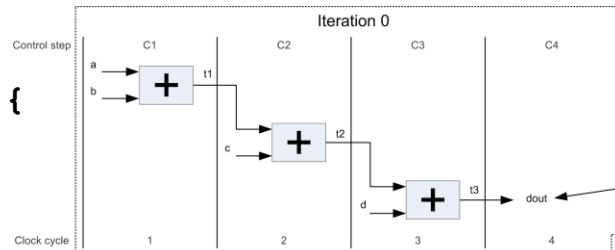
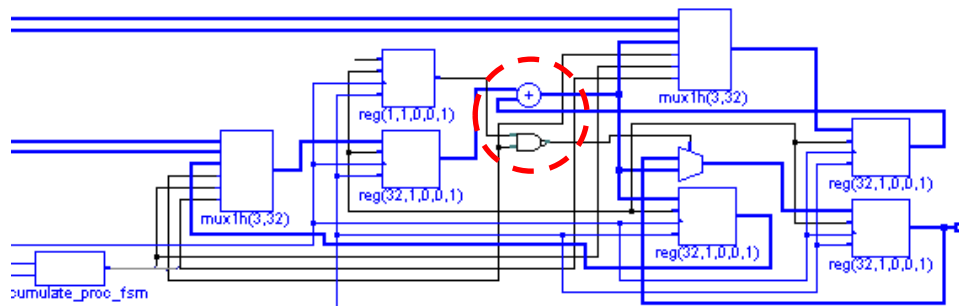
- Definitions:

- Initiation Interval (II) is how many clock cycles are taken before starting the next loop iteration
- Latency refers to the time, in clock cycles, from the first input to the first output
- Throughput, not to be confused with IO throughput, refers to how often, in clock cycles, a function call can complete

Loop Pipelining

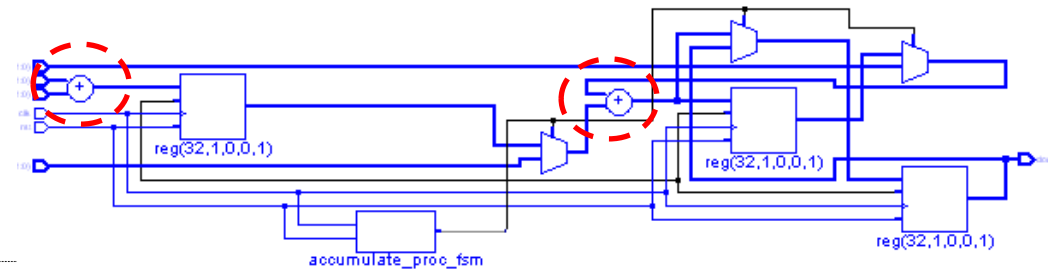
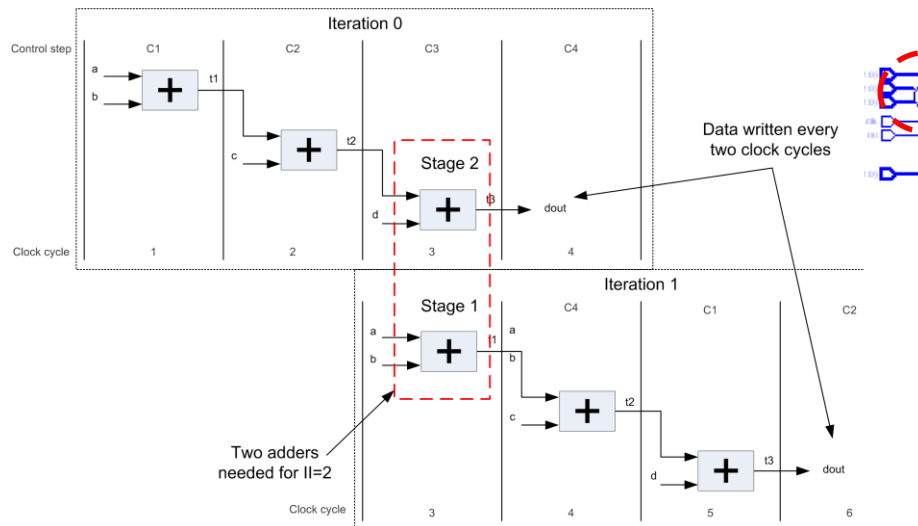
- Design is left unconstrained
 - Single pipeline stage because there's no overlap between execution of each iteration of the main loop
 - Results in data written every four clock cycles
 - Because there is no overlap of any operation only a single adder is required if sharing reduces overall area
 - Latency = 3, Throughput = 4

```
void accumulate(int a, int b,  
               int c, int d, int &dout){  
    int t1,t2;  
    t1  = a  + b;  
    t2  = t1 + c;  
    dout = t2 + d;  
}
```



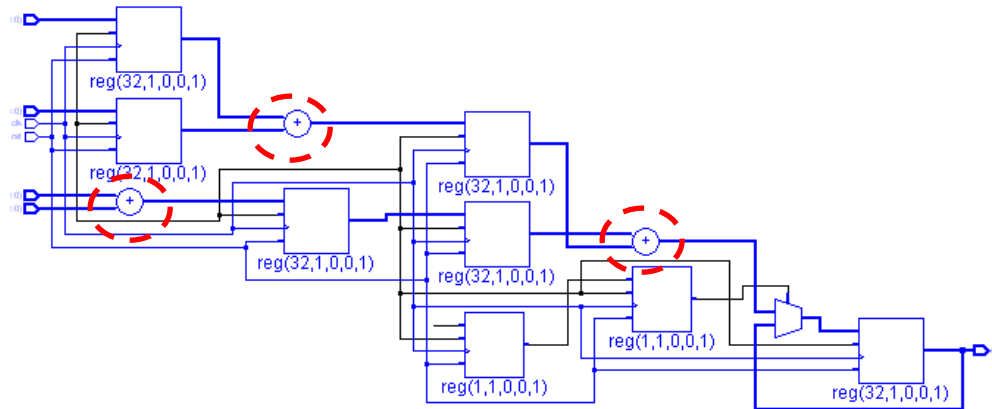
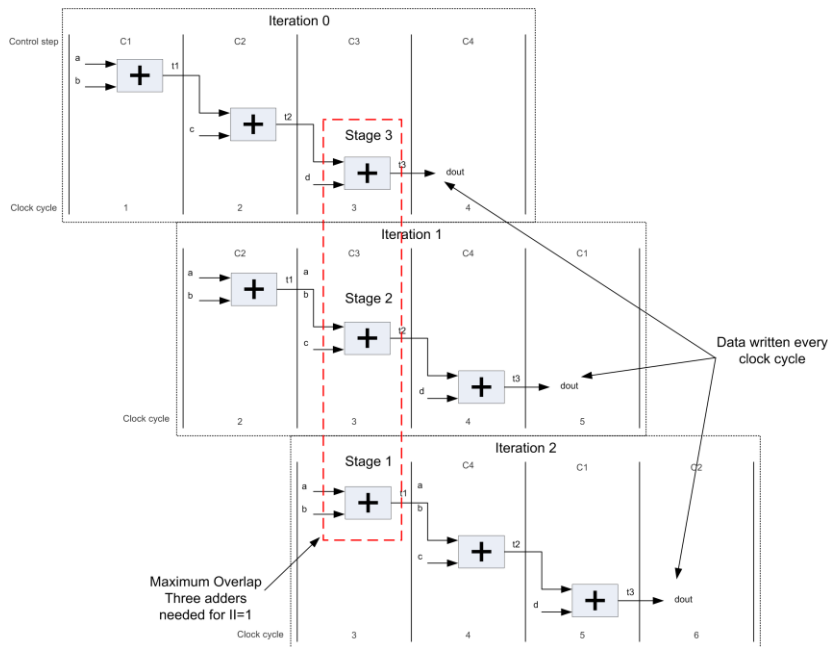
Loop Pipelining

- Pipelining the main loop with an $II=2$ results in a new iteration started every two clock cycles
 - Two pipeline stages
 - Iteration one is started in C3 while iteration 0 is computing " $t3 = t2 + d$ ".
 - Since iteration one is computing " $t1 = a + b$ " it can be seen that two adders are required for the two pipeline stages



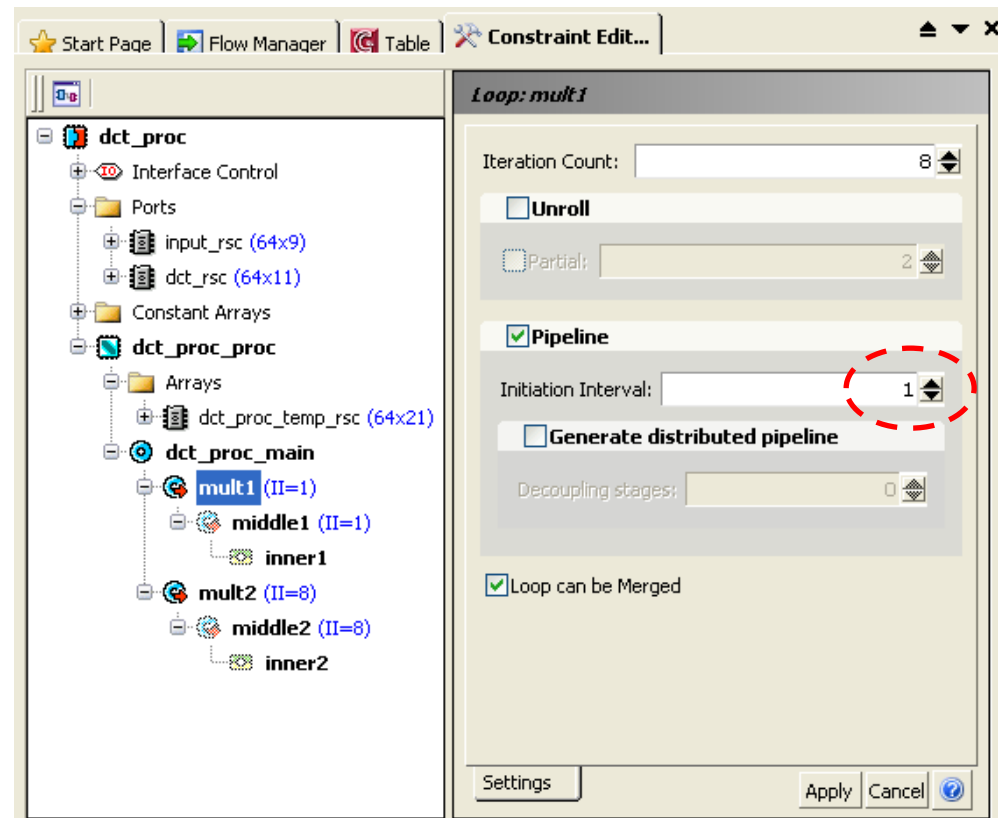
Loop Pipelining

- Pipelining the main loop with an $II=1$ results in a new iteration of started every clock cycle.
 - Three pipeline stages
 - Iteration one is started in C2 and iteration 2 is started in C3
 - Looking at C3 shows that three adders are required in hardware since all three pipeline stages are active



Loop Pipelining – Architectural Constraints

- The red arrow on the loop indicates that it is pipelined
 - Initiation Interval (II) is also shown
- Sub-loops are greyed out due to nested loop pipelining



When and How to Pipeline

- Use pipelining when you want to achieve a given throughput/data rate
- Use pipelining to reduce latency for iterative loops
- Use pipelining when full unrolling is not practical
- Always start with the innermost loops
 - Then work your way out
- Pipelining Initiation Interval can be increased to reduce area
- Nested loop pipelining can be used to improve system performance (throughput and latency)

Loop Merging

- Loop Merging implements two independent loops in parallel
- The resulting loop has iterations equal to the maximum of it's members

```
void loop_merging1 (unsigned char &mod,  
                    char a[3], char b[3],  
                    char c[4], char d[4]) {  
    for (int i = 0; i < 3; i++)  
        b[i] = (a[i] * mod) >> 8;  
    for (int j = 0; j < 4; j++)  
        d[j] = (c[j] * mod) >> 8;  
}
```

Two sequential Loops With Loop Merging Disabled

1	2	3	1	2	3	4
---	---	---	---	---	---	---

Two sequential Loops Merged

1	2	3	
1	2	3	4

Manual Loop Merging Code Example

- Merging happens automatically
- Two multipliers needed in merged version as both loops now run in parallel from a single index control

```
void loop_merging1 (unsigned char &mod,
                    char a[3], char b[3],
                    char c[4], char d[4]) {
    for (int i = 0; i < 3; i++)
        b[i] = (a[i] * mod) >> 8;
    for (int j = 0; j < 4; j++)
        d[j] = (c[j] * mod) >> 8;
}
```

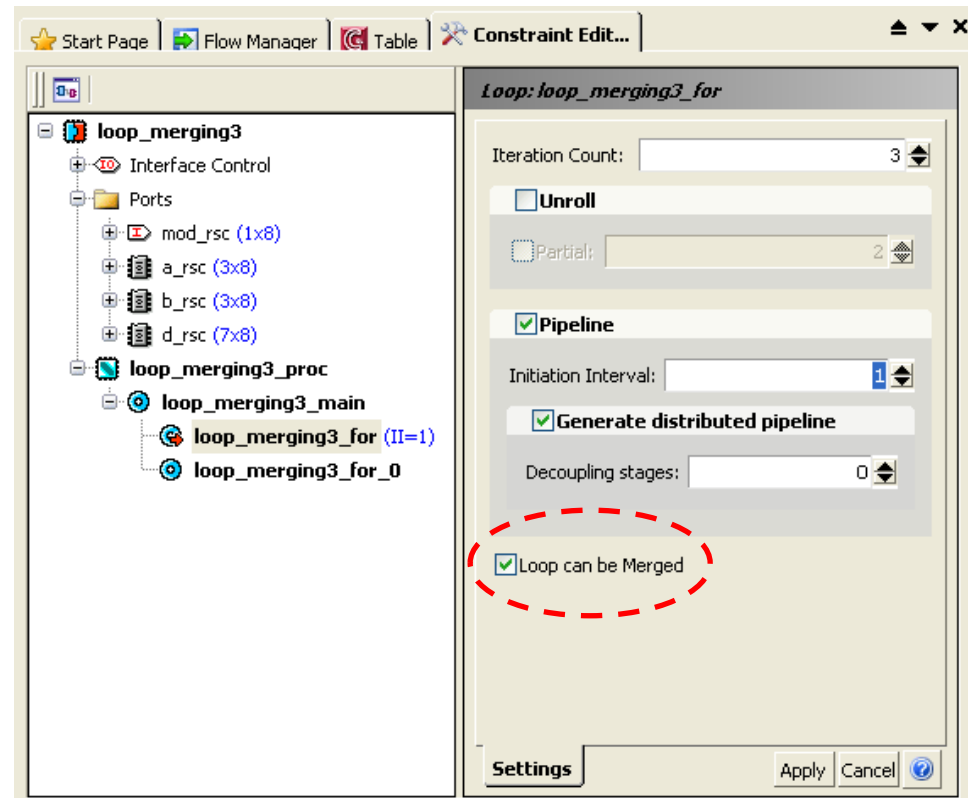


Equivalent
"manual"
loop
merging

```
void loop_merging2 (unsigned char &mod,
                    char a[3], char b[3],
                    char c[4], char d[4]) {
    for (int j = 0; j < 4; j++) {
        if (j < 3) b[j] = (a[j] * mod) >> 8;
        d[j] = (c[j] * mod) >> 8;
    }
}
```


Loop Merging Setting

- On by default for all loops
- Pipelining settings will only be saved from the loop with the smallest II
- Unknown iterations on a loop will cause resulting loop to have unknown iterations
- A fully unrolled loop will not be merged
- Memory & I/O accesses can affect if a loop can be merged or what the outcome will be



When to Disable Loop Merging

- Loop merging is enabled by default
 - It can be disabled globally or for an individual loop
- I/O accesses may require that two loops run sequentially
- Merging may prevent the sharing of operators or components across multiple loops

Loop Optimization Summary

- Be methodical in order to converge on a good solution
 - Use Catapult analysis and design goals to determine & set constraints
 - Always start from the inner loops and work your way out
- Loop unrolling controls parallelism
- Loop pipelining controls throughput

Agenda

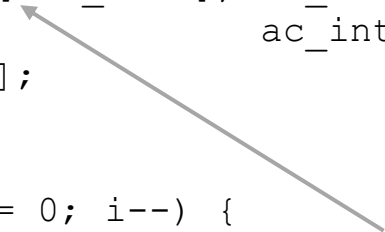
- Loops and C++ style
- Loop Optimizations
 - Unrolling
 - Pipelining
 - Merging
- Loops and Memories
- Pipeline Feedback Analysis

Unrolling Loops Containing Memories

- Arrays mapped to memories tend to be the performance bottleneck in a design
 - Memory bandwidth limits throughput/parallelism
 - Memory word-width and unrolling can be combined to increase performance
- Consider the following (FIR filter) design:

```
#pragma design top
void fir_filter (ac_int<8> coeffs[NUM_TAPS], ac_int<8> *input,
               ac_int<8> *output ) {

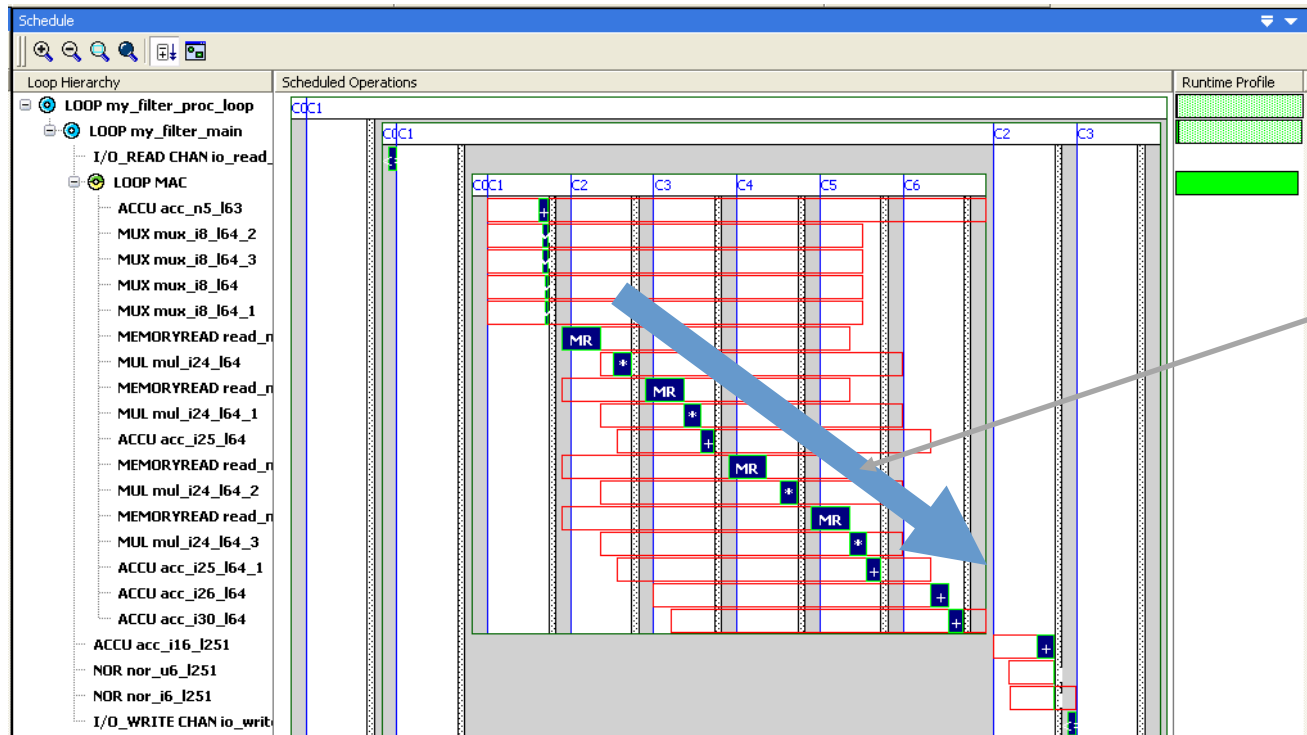
    static ac_int<8> regs[NUM_TAPS];
    int temp = 0;
    int i;
    SHIFT:for (i = NUM_TAPS-1; i >= 0; i--) {
        if (i == 0)
            regs[i] = *input;
        else
            regs[i] = regs[i-1];
    }
    MAC:for (i = NUM_TAPS-1; i >= 0; i--) {
        temp += coeffs[i]*regs[i];
    }
}
```



“coeffs” array mapped to a singleport RAM

Loop Unrolling & Scheduling

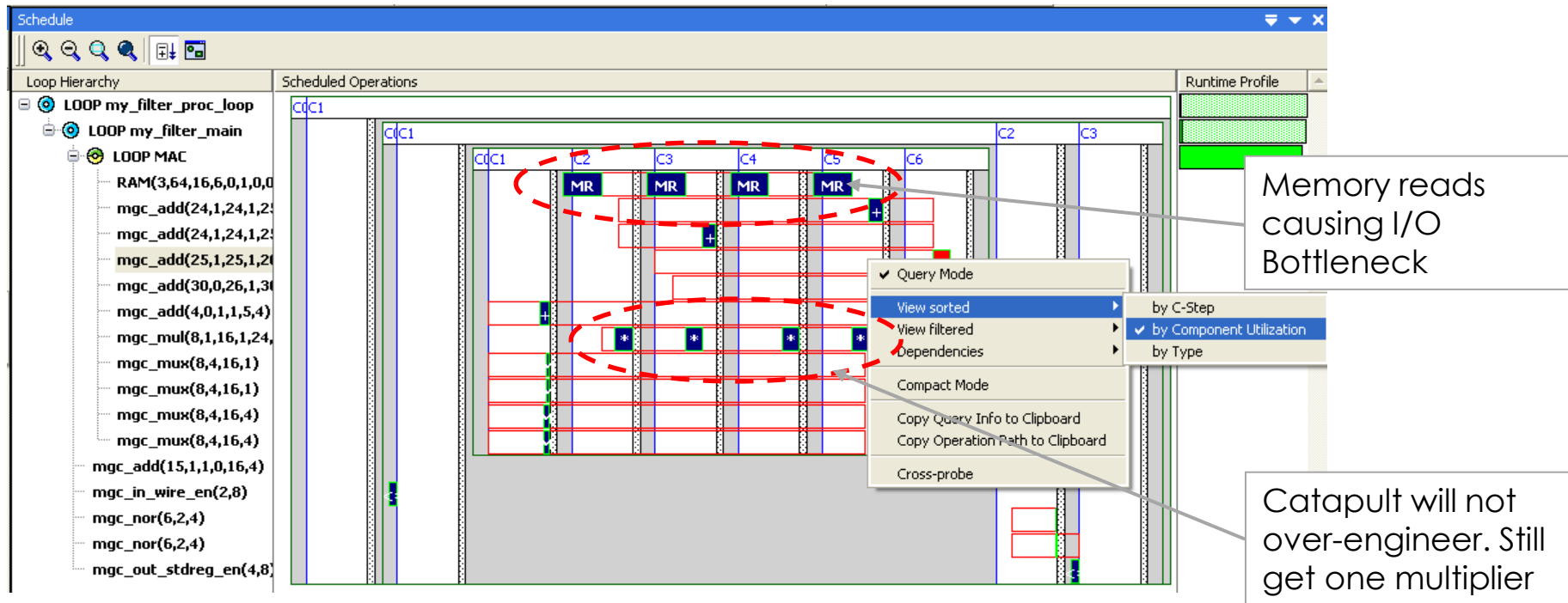
- Coefficients stored in single port RAM
- Unrolling MAC loop partially by 4
 - Need four RAM reads per “new MAC” loop iteration
 - Bandwidth limited => No improvement in performance



Diagonal schedule indicates lack of parallelism. Often due to bandwidth limitation or other dependencies

Loop Unrolling, Pipelining & Scheduling

- Use “View sorted => by Component Utilization”
- Cannot pipeline $ll=1$ (only $ll \geq 4$)



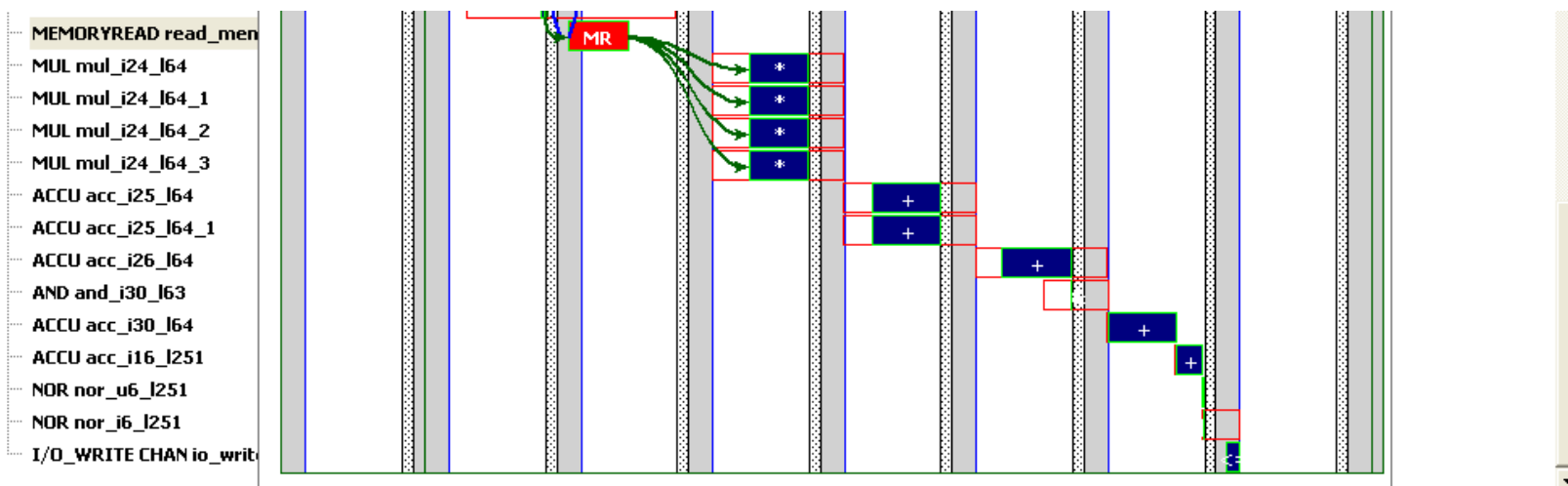
```
# Error: insufficient resources 'ram_sample-090nm-singleport_beh_dc.RAM(3,64,16,6,0,1,0,0,0,1,1,64,16,1)' to  
# schedule 'my_filter_proc'. 4 are needed, but only 1 instances are available (SCHD-4)  
# Error: Design 'my_filter' could not schedule partition '/my_filter/my_filter_proc' - resource competition
```

Match Loop Unrolling and I/O Bandwidth

- Optimum implementation is when bandwidth and loop unrolling are matched, with pipelining $II=1$
- In this example, 4 memory reads are needed per iteration of the partially unrolled “new MAC” loop
- Several options are available:
 - Widen Coefficient RAM by 4x
 - Use Dual Port with width 2x
 - Four Single Port RAM's Interleaved
 - Two Single Port Ram's Interleaved using 2x width
 - Wire interface
- No need to change C++ code
 - Pick the interface and implementation you want
 - Use constraints to set the I/O or memory architecture

Widen Coefficient RAM 4x

- One memory read reads 4 coefficients
 - Single memory read shown in Gantt chart
- Additional operator logic required to perform arithmetic in parallel
 - 4 multipliers
 - 3 adders for summing of 4 multiply results (adder tree)
 - 1 adder with feedback for accumulator
 - 1 adder for rounding logic



Lab 6 – Loop Optimizations

- This lab exercise covers:
 - Constraining and Scheduling of two FIR filters
 - Standard FIR
 - Folded FIR
 - Interface Synthesis
 - Loop unrolling and pipelining
 - Automatic Loop merging
 - Area/Latency goals
 - Output Report and analysis
- Unzip ../lab6.zip
- Go to the Lab6 directory
- Follow the instructions in Lab6.doc

CALYPTO[®]

Design • Optimize • Verify