HLS Presentation

Angelos
Athanasiadis

Outline

- Vivado is a big system
  - UG902 – This is the user's guide
    - It is > 700 pages (lots of pictures, but not meant for skimming)
  - UG871 – Tutorial Guide
- Impossible to cover in 1 hour → take the 20,000 foot view of the
  - Development process
  - Refinement process
    - Time optimization
    - Resource optimization
- Focus more on the *What* can be done rather than the *How*
- Go through a simple example
- If you retain as much as, "*Oh, I know you can do something like that*", it will have served some purpose

Development Process

- Vivado HLS is an Eclipse based IDE
  - This allows you to get going quickly
  - There are ways to script the development process
- You break your code into 2 pieces
  - A test harness
    - This runs only on the host
  - One *top-level* procedure
    - This is the code eventually destined for the FPGA, but
    - Only after you debug and simulate on a friendly host

Development Process

- The test harness provides test vectors to the FPGA destined code
- The initial development and testing is completely host-based in 3 steps
  - No FPGA/hardware is necessary
- Step 1. C-Simulator simulates the FPGA using strictly C-code – < minutes
  - A fast edit/compile/link/test cycle
- Step 2. Synthesis stage – ~10 seconds - 10 minutes
  - Produces the VHDL (or Verilog)
  - This gives good (but not perfect) timing and resource usage
- Step 3. Can now run an analysis and co-simulator on this VHDL/Verilog
  - The analysis produces accurate resource usage
  - The co-simulator produces detailed timing (waveform)
  - Both the analysis and co-simulation are much slower
- Final step is producing a downloadable bit file – ~hours

What it does

- Vivado HLS allows one to write algorithms in
  - C/C++
  - System C.
  - OpenCL seems to working itself into the mix
  - Would recommend stick to C++
    - Looks like the best supported

- Just throwing vanilla C/C++ at Vivado HLS will not work
  - These are sequential languages
    - FPGAs get their power from parallelism
  - FPGAs are not constrained to natural 8/16/32/64 - bit boundaries
    - Any size integer or fixed point are possible
  - Some constructs natural to an FPGA have no counterparts in C/C++
    - *e.g.* multi-port memory
  - C/C++ is like a visitor in a foreign country
    - They may speak the language, but do not appreciate the culture
    - Your job → Absorb/understand the culture,
    - Vivado's role → Help you in bridging this cultural gap

Decorated C++

How to bridge the gap

- Two tools are
  - Language augmentations
  - Pragmas
- Language augmentations
  - These are C++ classes during the simulation stage, then…
  - Mapped to specific hardware constructs during synthesis
  - Most common examples are arbitrary precision classes
    - *e.g.* ap_uint<12>
    - Easier in C++ than C because other classes (like printing) understand them
    - Advise using typedef's to make these easy to change
      - typedef ap_uint<12> Adc;

Decorated C++

Bridging the Gap - Pragmas

- Pragmas, a very large topic
  - Allow creation of multi-port memories
  - Loop unrolling
  - Pipelining
  - Interface specification
  - Array partitioning
  - Array reshaping
  - Dataflow
  - Resource control
  - …and way more than can be covered
- Gaining an understanding of their usage is a key component to success

Some Fine Print

- The language is C/C++, but the target is an FPGA
  - Algorithms and styles that work in a sequential machines may or may not translate
- Currently,
  - A clear leaning towards pipeline style processing
  - This may just reflect traditional FPGA applications
- Buffering and decimation are trickier
  - Xilinx seems to have realized this
  - Better tools/techniques to deal seem to be coming

Even Finer Print

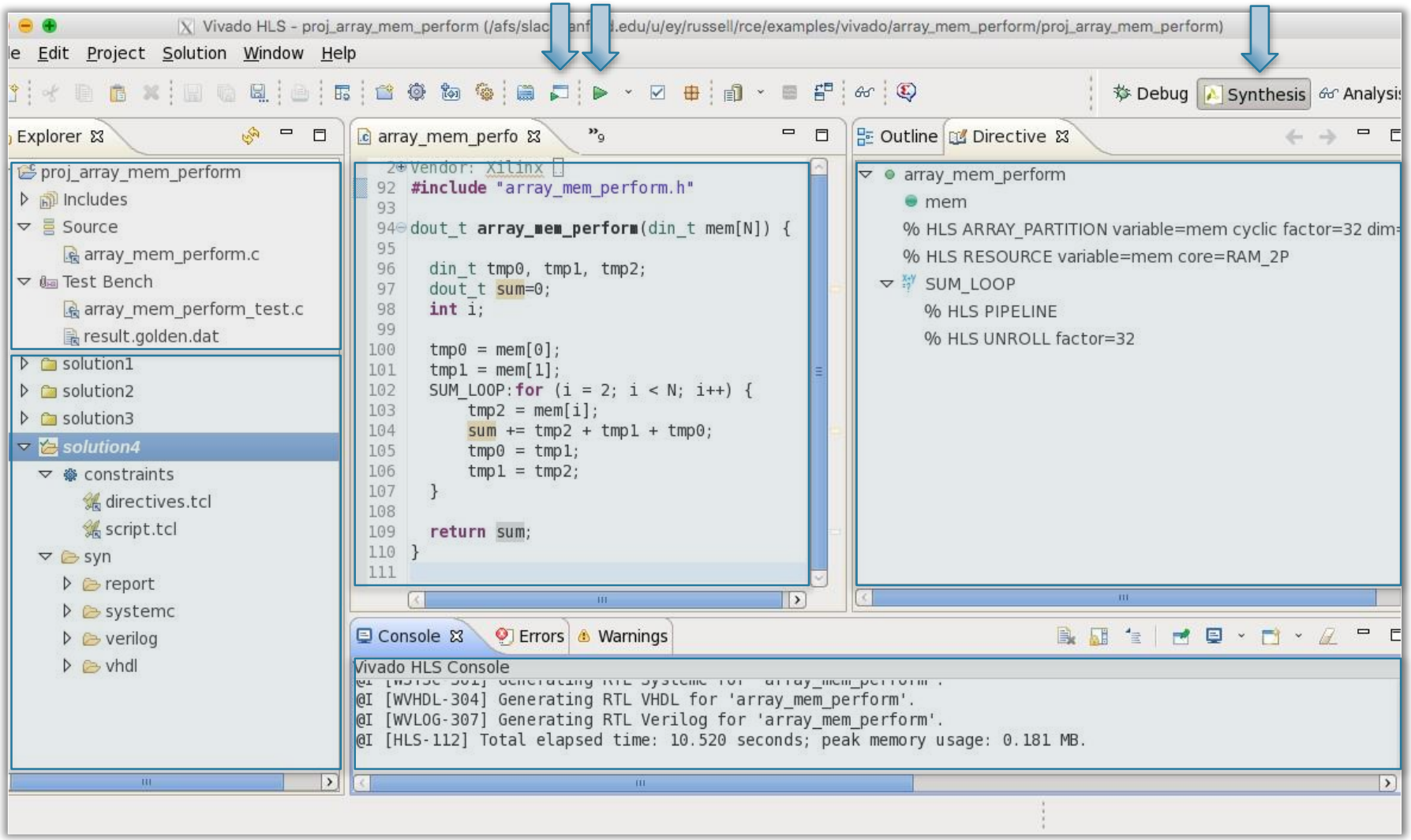
- More suited to *algorithmic* code, not the IO
  - Depend on VHDL to handle decoding of raw bit streams
  - Currently depend on VHDL to do the DMA to the processor
    - This may be relieved in SDSoc – but not for the raw input bit streams
    - Locally we refer to this as *coding in the donut hole*
- Have had issues dealing with large codes
  - Had to break the waveform extraction code handling 128 channels in 4 x 32 code blocks
    - May have learned, current DUNE compression code handles 256 channels
      - Synthesis ~ 150 seconds
      - Export (with analysis) ~ 30 minutes
      - Haven't built a viable bit-file yet, nothing to report here
- Model of 1 test harness and 1 FPGA destined module is limiting
  - In the waveform extraction code, would have like to have a 2nd module that recombined the 4 x 32 output streams.
  - SDSoc may be addressing this

Example of Code Development

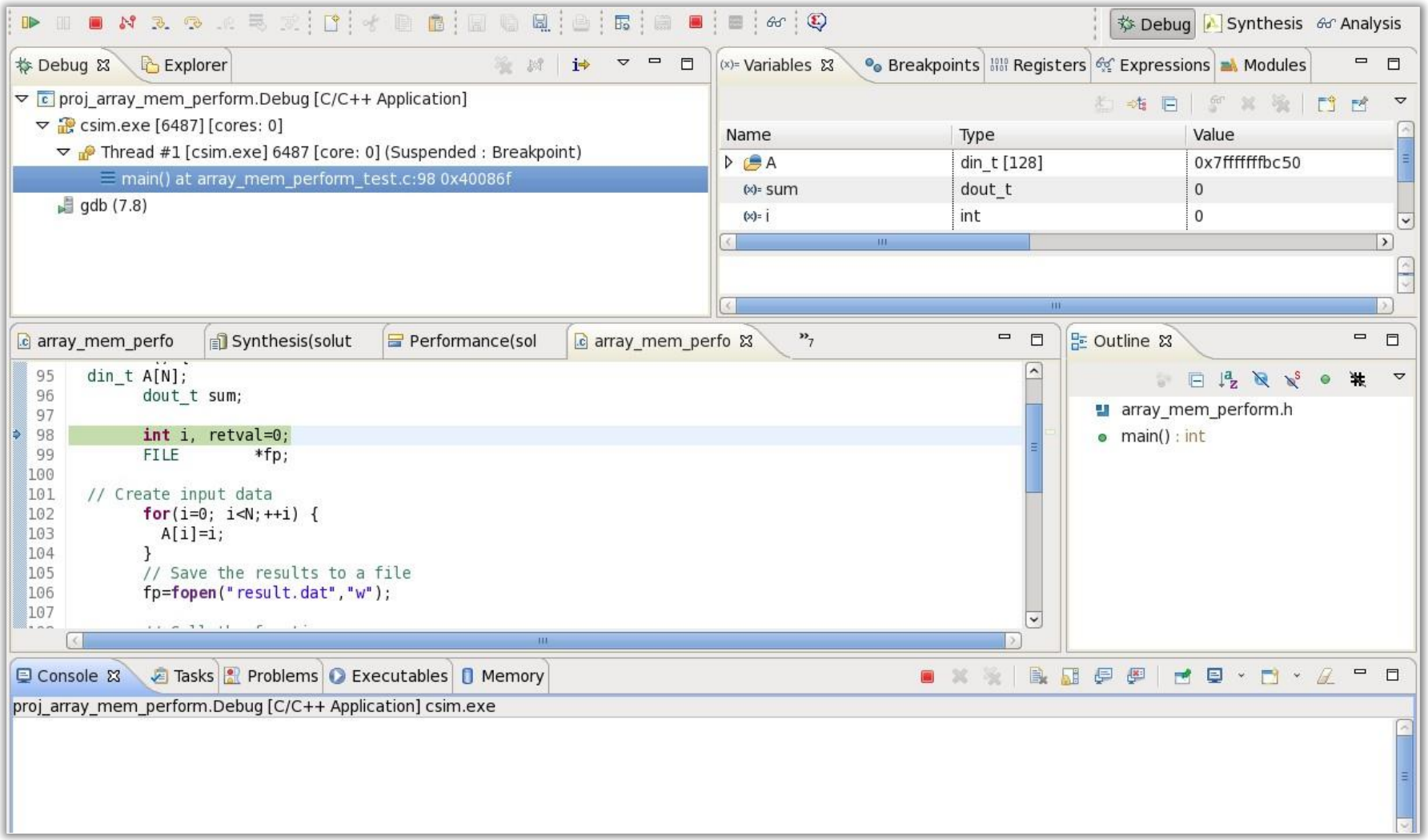
- Will use a very simple example to illustrate the process.
- The general cycle is
  - Write the test harness and top level code
  - Compile and debug it
  - Synthesis it to see where the time and resources are going
  - Adjust the code
  - Add pragmas
- Will largely ignore the first two steps
- Emphasis again
  - You never leave the comfort of your host machine during these steps

But First…

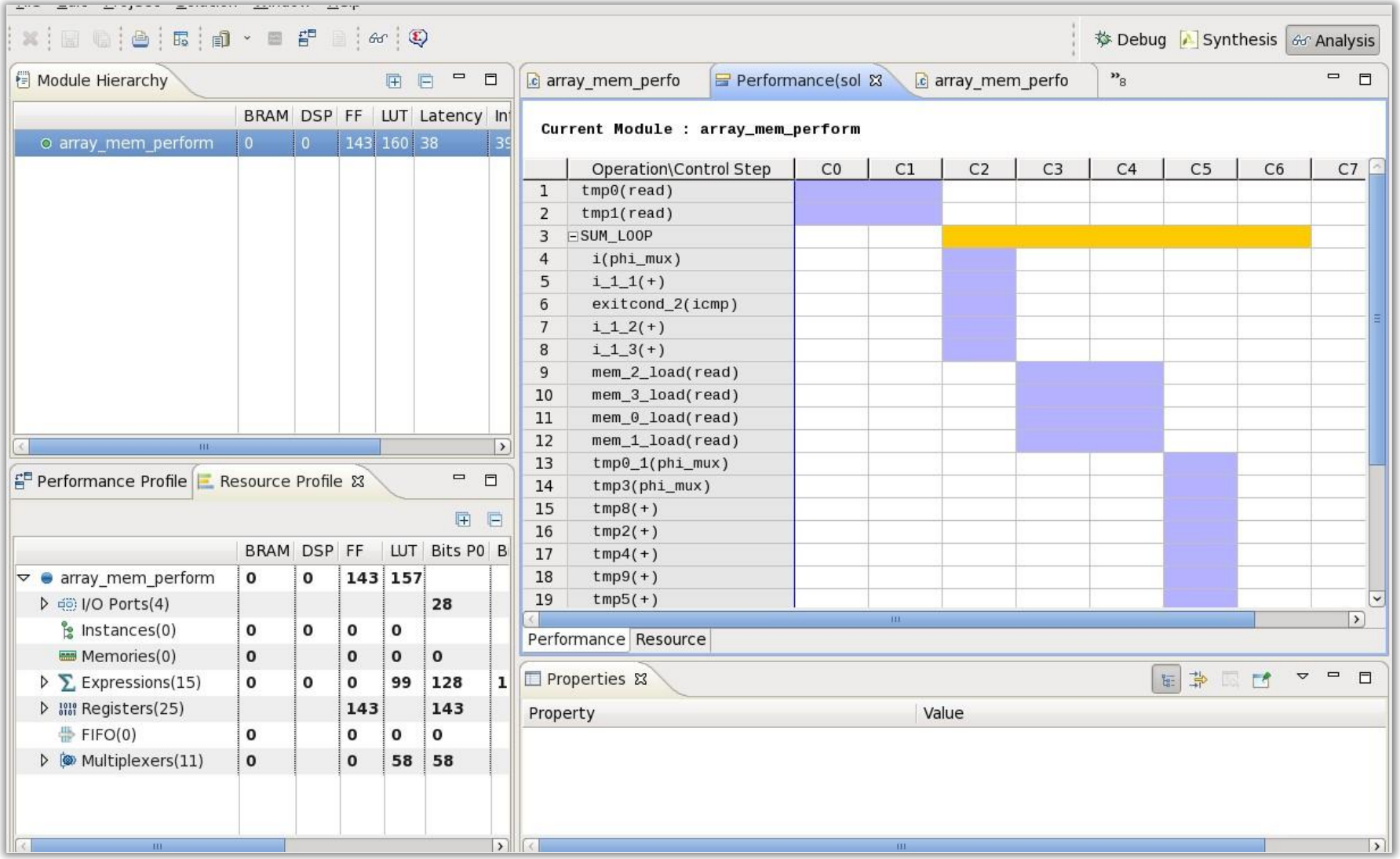
The Anatomy of the IDE



Synthesis View


Debug
Synthesis
Analysis
Debug
Explorer
proj_array_mem_perform.Debug [C/C++ Application]
csim.exe [6487] [cores: 0]
Thread #1 [csim.exe] 6487 [core: 0] (Suspended : Breakpoint)
main() at array_mem_perform_test.c:98 0x40086f
gdb (7.8)
Variables
Breakpoints
Registers
Expressions
Modules
Name
Type
Value
A
din_t [128]
0x7fffffffbc50
sum
dout_t
0
i
int
0
array_mem_perfo
Synthesis(solut
Performance(sol
array_mem_perfo
din_t A[N];
dout_t sum;
int i, retval=0;
FILE *fp;
// Create input data
for(i=0; i<N;++i) {
A[i]=i;
}
// Save the results to a file
fp=fopen("result.dat","w");
Outline
array_mem_perform.h
main() : int
Console
Tasks
Problems
Executables
Memory
proj_array_mem_perform.Debug [C/C++ Application] csim.exe

Debug View



Analysis View

Simple Example

- The example is from the Vivado Example area
  - Would encourage you to look there
  - These are simple examples
    - Just illustrate a particular aspect or technique
    - They are available off the initial welcome screen
- The example merely sums the elements of an array
  - Will serve as a way to
    - Navigate through the myriad of displays
    - Demonstrate a couple of common techniques

Memory Bottleneck

```
dout_t array_mem_bottleneck(din_t mem[N]) ←Note the use of types
{                                                (N = 128)
  dout_t sum=0;

  SUM_LOOP: for(int i=2;i<N;++i)   ← Note the label, this is how one
  {                                   scopes pragmas
    sum += mem[i];                 ← Asking for 3 memory references
    sum += mem[i-1];                  on each iteration. This creates
    sum += mem[i-2];                  a memory access bottleneck
  }

  return sum;
}
```

Bottleneck

- Poor performance
  - ~2 cycles per iteration
  - The goal is usually 1 cycle

Summary

Latency		Interval		
min	max	min	max	Type
254	254	255	255	none

Detail

Instance

Loop

- Note the resource usage

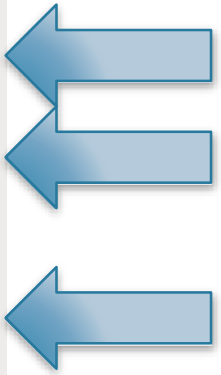
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	54
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	34
Register	-	-	55	-
Total	0	0	55	88
Available	650	600	202800	101400

From Analysis View

	Operation\Control Step	C0	C1	C2	C3
1	⊟SUM_LOOP				
2	sum(phi_mux)				
3	i(phi_mux)				
4	exitcond(icmp)				
5	mem_load(read)				
6	tmp_2(+)				
7	mem_load_1(read)				
8	tmp_5(+)				
9	i_1(+)				
10	mem_load_2(read)				
11	tmp1(+)				
12	tmp_1(+)				
13	sum_1(+)				



Better Code

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

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

  tmp0 = mem[0];                          ← Move 2 of the references
  tmp1 = mem[1];                            out of the loop
  SUM_LOOP:for (int i = 2; i < N; i++) {
     tmp2 = mem[i];                       ← Now, only 1 memory reference
     sum += tmp2 + tmp1 + tmp0;             per iteration
     tmp0 = tmp1;
     tmp1 = tmp2;
  }

  return sum;
}
```

Better Code
→ Better Performance

- Improved performance
  - →1 cycle per iteration
    - The extra cycles are loop entrance and exit latency

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
132	132	133	133	none

Detail

Instance

Loop

- Resource Usage has barely changed
  - Up by 1 LUT
- This is a good trade off

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	38
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	50
Register	-	-	72	1
Total	0	0	72	89

Pragmas

Overview

- To further improve performance, need to help Vivado out by using pragmas
- There are many, many pragmas and lots of variations for any given pragma
- You can restrict the scope of a pragma
  - Functions
  - Loops
  - Regions
  - There are a few exceptions, like PIPELINE which applies all the way down a hierarchy

Pragmas

How to specify

- Specification of pragmas can be either
  - Directly in the code
    - This is appropriate for
      - Those unlikely to change, *e.g.* pragmas defining the interface
      - Code to be released
  - In *named* solutions
    - This is information (think include files) that are kept separate from the code, but selectively applied to it
    - Can be any number of solutions; with multiple solutions
      - You can play *What if* games without hacking the source code.
      - Define solutions for different target FPGAs
    - You select one of the solutions when you synthesis

Pragmas
Uses

- There are 2 main uses
  - Improve performance
  - Control resource usage
- While some pragmas are directly aimed at one or the other of these
  - There are some (ARRAY_RESHAPE) that address both
- There is a third use
  - These attempt to make the diagnostic information more useful
  - They do not affect the generated code
    - *e.g.* TRIPCOUNT can be used to specify a *min,max and average* count on variable iteration loops
      - This helps make the timing more meaningful
- And yet a fourth use
  - These help when Vivado is unable to correctly infer properties
    - *e.g.* DEPENDENCY can be used to express or negate a variable dependency

Popular Pragmas

RESOURCE

- Can be used to specify details of the memory
  - Memory can be implemented in
    - Block Ram (BRAM)
    - LUT (LUTRAM)
  - It can be any of
    - RAM
    - ROM
    - STREAM
    - FIFO
  - It may be (where it makes sense)
    - Single ported
    - Couple of different styles of dual porting
- Example
  - #pragma HLS RESOURCE variable=arr core=RAM_2P_BRAM
- *Caveat*, this pragma does a lot more than this

Popular Pragmas

ARRAY_PARTITION

- Adds ports to memory
  - This can relieve memory bottlenecks
  - Almost always needed when trying to achieve parallelism
- The ports can be added differently to different dimensions of multi-dimensional arrays
- They can have 1 of 3 styles
  - Complete
  - Cyclic ← This is the most common
  - Block
- Example
  - #pragma ARRAY_PARTITION variable=d2 dim=2 cyclic factor=4

Popular Pragmas

DATAFLOW

- The DATAFLOW pragma allows 2 or more functions/loops/regions to execute in parallel
  - Think of it as analogous to multi-threading
- Useful in packetized processing, *e.g.*
  - *Read in a packet → Process It → Write it out*
- Unlike multi-threading, overhead is not an issue
  - This can be used at very small granularity
- *Caveat* – DATAFLOW, PIPELINE and UNROLL come with lots of *terms and conditions*
  - These must be understood to be used effectively
    - No just clicking on the *I agree* box

Popular Pragmas

INLINE,PIPELINE,UNROLL

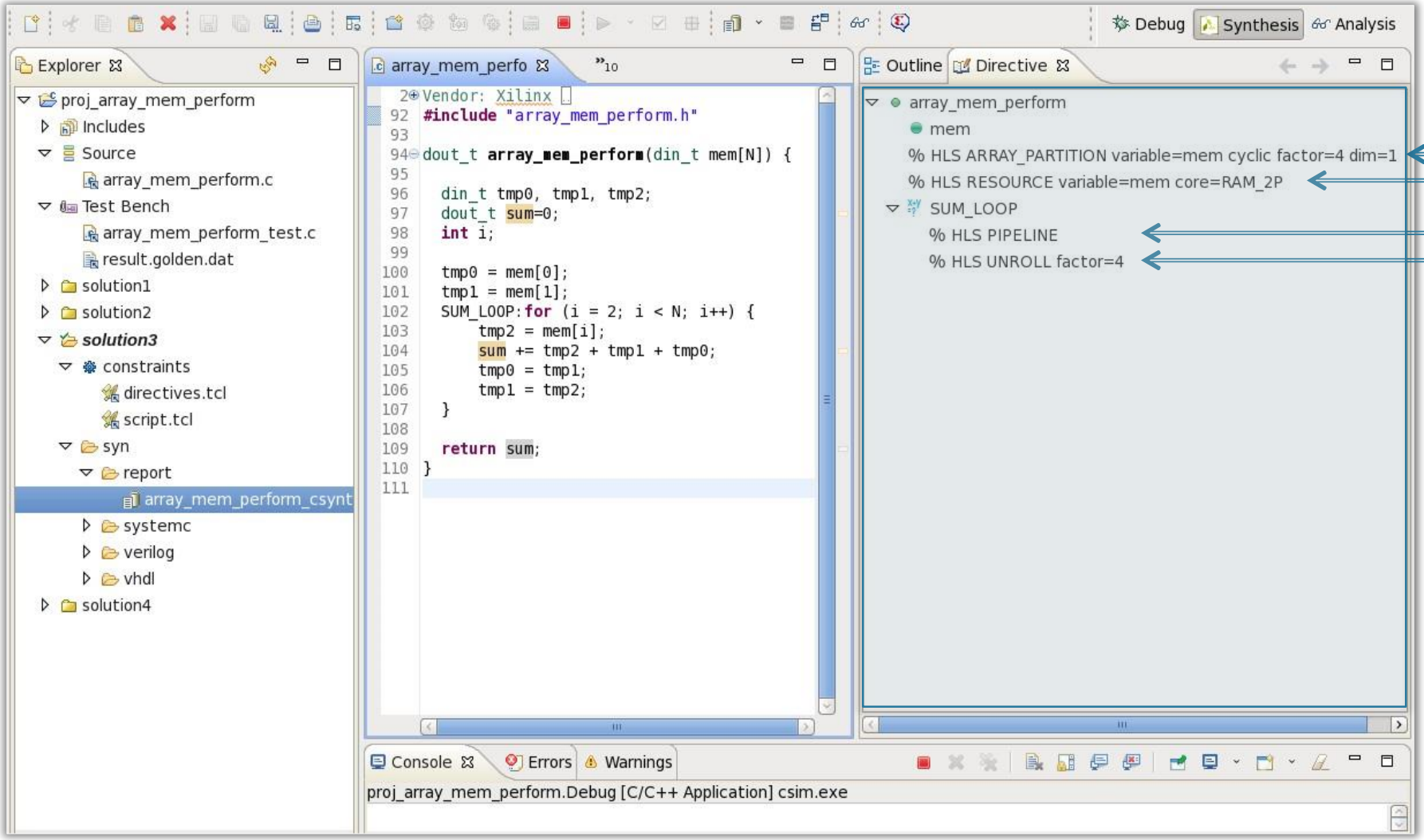
- Functions can be *inlined*
  - This can help in some cases / hurt in others
  - Example
    - #pragma HLS INLINE (off)
- Functions, loops can be PIPELINED
  - Allows these to accept more input as soon as they are able
  - Example
    - #pragma HLS PIPELINE
- Loop unrolling
  - Determines the extent to which a loop will be unrolled (or not)
  - Example
    - #pragma HLS UNROLL factor=4

Language Augmentations

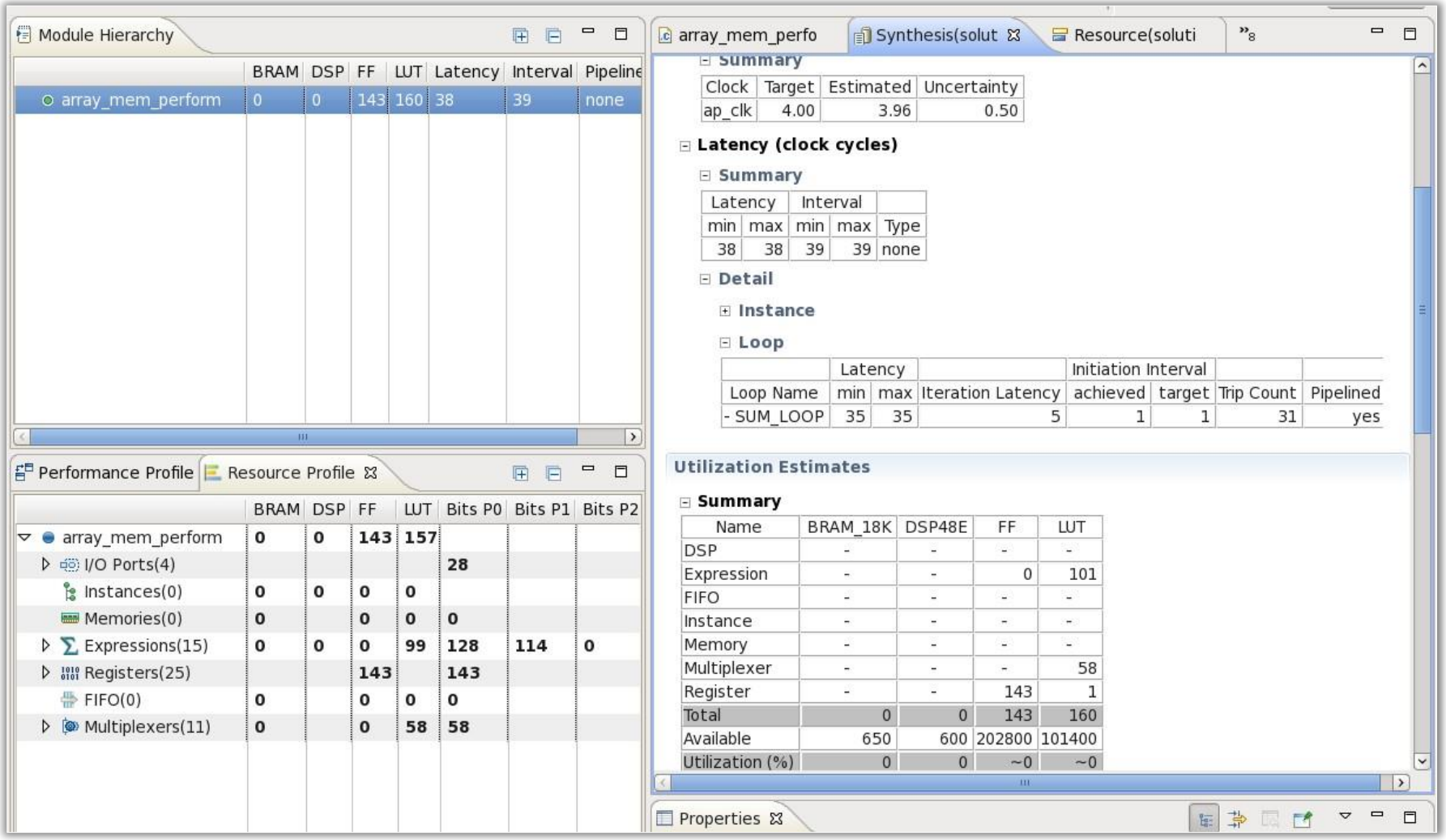
- These are C++ classes that map onto hardware constructs
- Examples are
  - ap_int<n>, ap_uint<n> - arbitrary precision integer
    - These have many bit related methods associated with them
      - Bit extract
      - Bit concatenation
      - Bit reversal
    - These are heavily used
      - The appropriate width improves time and resource usage
    - Also can specify fixed point types
    - Strongly encourage that these are captured in typedef's.
      - Make that more than strongly → *Do it!*
  - hls_stream – stream variables
  - ap_fifo – fifo variables

Exploring the Effect of Pragmas

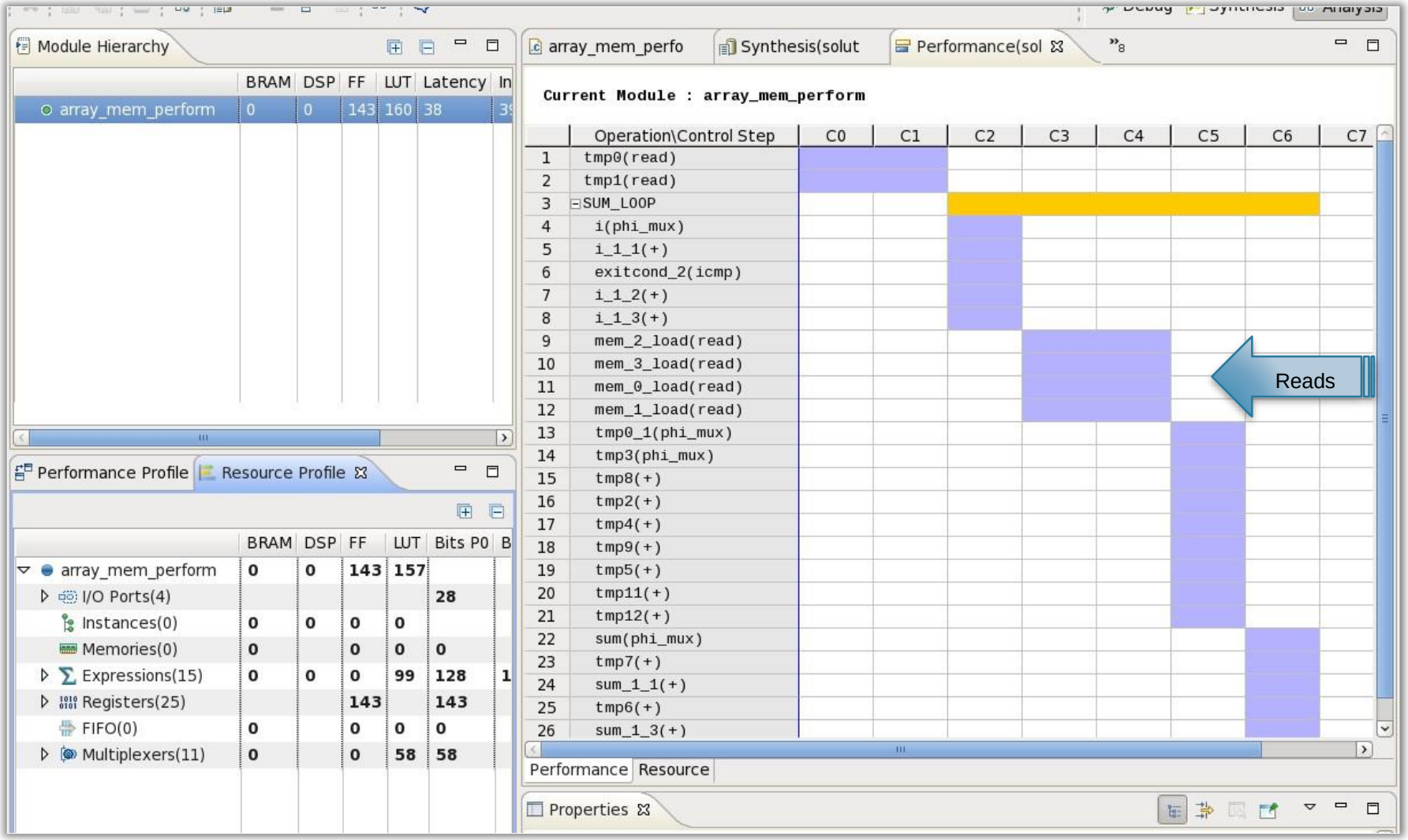
- The next few slides show what happens when the code is tweaked with appropriate pragmas.
- This simple piece of code can be vastly improved
- Of course there is a cost to be paid, so watch both
  - The time
  - The resource usage



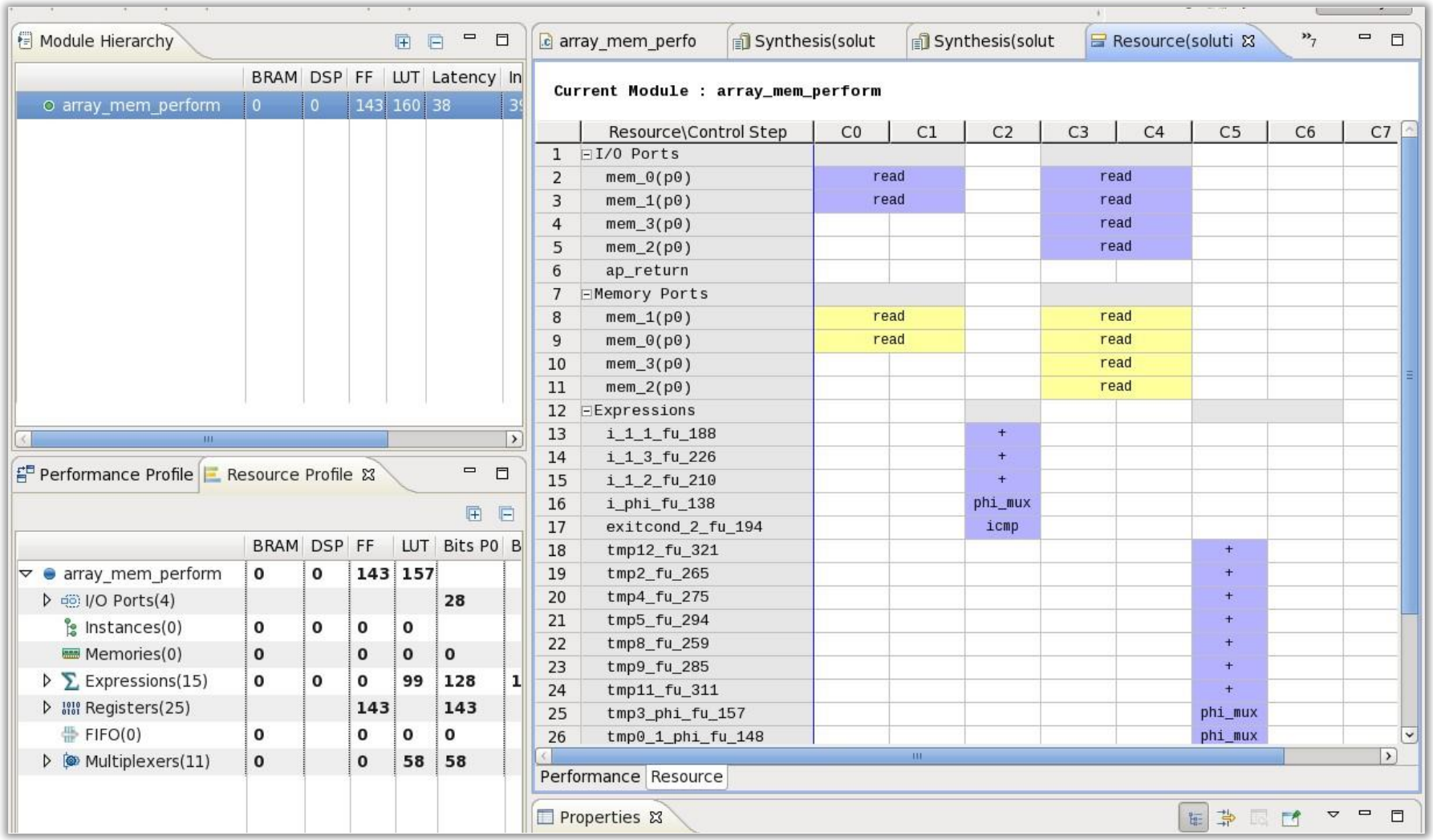
Using Pragmas:: Synthesis View



Analysis Summary View



Analysis Performance View



Analysis View - Resource

Performance Comparison

- Can compare the effects of the different solutions
- Timing comparisons

Latency (clock cycles)

		solution1	solution2	solution3	solution4
Latency	min	132	132	38	11
	max	132	132	38	11
Interval	min	133	133	39	12
	max	133	133	39	12

- Resource comparisons

Utilization Estimates

	solution1	solution2	solution3	solution4
BRAM_18K	0	0	0	0
DSP48E	0	0	0	0
FF	72	158	143	709
LUT	89	161	160	958

Just to show you just can't turn knobs

The following solutions increase the loop unrolling x2 each time.

Solution 7 – Unroll x 16

Solution 8 – Unroll x 32

Solution 10 - Complete

Performance Estimates

Timing (ns)

Clock		solution4	solution5	solution6	solution7	solution8	solution9	solution10
ap_clk	Target	4.00	4.00	4.00	4.00	4.00	4.00	4.00
	Estimated	3.96	5.94	6.93	7.92	8.91	3.38	3.38

Latency (clock cycles)

		solution4	solution5	solution6	solution7	solution8	solution9	solution10
Latency	min	38	23	15	11	10	5	68
	max	38	23	15	11	10	5	68
Interval	min	39	24	16	12	11	6	69
	max	39	24	16	12	11	6	69

Utilization Estimates

	solution4	solution5	solution6	solution7	solution8	solution9	solution10
BRAM_18K	0	0	0	0	0	0	0
DSP48E	0	0	0	0	0	0	0
FF	143	268	440	738	1406	2214	2284
LUT	160	275	504	967	1905	2717	3192

Observations

- Vivado HLS can be finicky
  - Sometimes it does what you want/expect
  - Other times, you wind up with a puzzled look
- It can be unstable
  - Seemingly innocuous changes can led to large changes in time and resource usage
    - I've adopted the *never make more than 1 change at a time* rule
- Development is largely a 1 or 2 person activity
  - Not exclusively a property of Vivado HLS, but it is contributor
  - Stems from the dedicated way FPGAs are used
    - Not like a CPU where you have multiple processes and tasks running that come from a cadre of developers

Going Forward

- Xilinx is betting on Vivado HLS to make FPGAs into a viable choice
  - It is not just the skill set – *i.e.* more C/C++ than VHDL coders
  - It is the complexity and size of what you want to do
    - Will overwhelm you and become unmaintainable
    - This akin to coding in assembler vs C/C++
      - Some things are appropriate to do in assembler, but…
      - No way all of it can be in assembler
  - Portability of moving to different FPGAs
- For those of us using the RCE,
  - The FPGA is where the power of the RCE lies
- While there is no free lunch,
  - This may make it somewhat cheaper