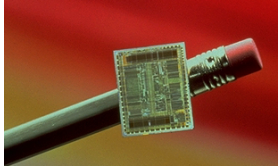
TELECOM Paris

IP PARIS



ICS901 / A2HW

HLS: High-Level Synthesis

Alaa Eddine MAZOUZ

alaa.mazouz@telecom-paris.fr

Based on the original slides by Yves Mathieu

Institut Mines-Télécom

Outline

Introduction: what HLS is and how it works

Principles: one example in four steps

Latency, II and pipelining

Loop unrolling

A real loop, and the hints that steer the tool

Limits and choices

Interfaces and practice

The question of this course: given a C++ function and a constraint, which hardware should we build, and how do we know?

The designer's dream

From RTL synthesis to HLS synthesis

RTL: you write

- every register
- every clock cycle
- every bit width
- and again for each new architecture

HLS: you describe

- the algorithm
- the target: ASIC xxx, FPGA yyy
- the clock frequency
- the throughput: calls per second
- the latency: cycles from a call to its result

High-Level Synthesis: the tool chooses the operators, the registers and the clock cycles.

HLS methodologies and tools

Languages and pragmas

Open questions

- No precise definition of a high-level description of hardware
- The synthesisable subset of a language depends on the tool: no dynamic memory, no recursion, loops with a known number of iterations
- How to model buses and protocols?

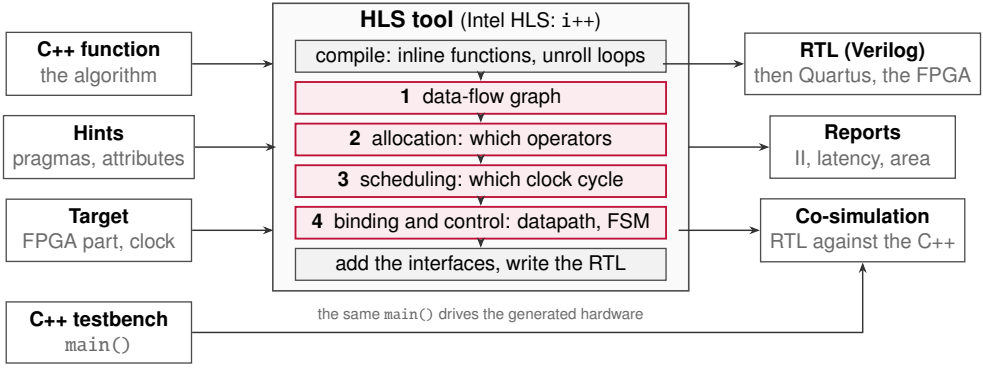
What every tool does

- A compiler that generates RTL instead of object code
- *Pragmas* carry the designer's intentions: parallelism, pipelining, RAM or flip-flops
- No standard: each tool has its own pragmas

GCC manual: "The `#pragma` directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself."

How HLS works: the big picture

What goes in, what the tool does, what comes out

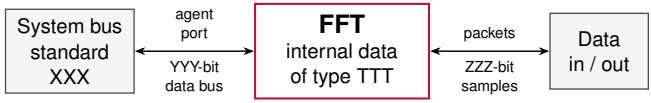


Steps 1 to 4 (in red) are the heart of every HLS tool. The next part follows them on one small example.

What question do we want to solve?

The real problem

Not “I want to code an FFT”, but “I want an FFT *that fits my system*”.



TTT: floating point, fixed point, integer...

Where RTL time goes

- control state machines
- communication with the outside: ad hoc protocols, standard buses

HLS saves most of its time on control and interfaces, not on the arithmetic.

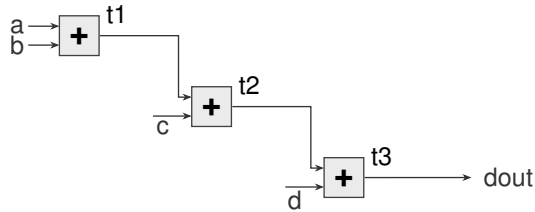
Outline

Introduction: what HLS is and how it works
Principles: one example in four steps
Latency, II and pipelining
Loop unrolling
A real loop, and the hints that steer the tool
Limits and choices
Interfaces and practice

Step 1: data flow graph

One example, followed through every step

```
#include "accum.h"
void accumulate(int a, int b, int c,
                int d, int &dout)
{
  int t1, t2;
  t1 = a + b;
  t2 = t1 + c;
  dout = t2 + d;
}
```



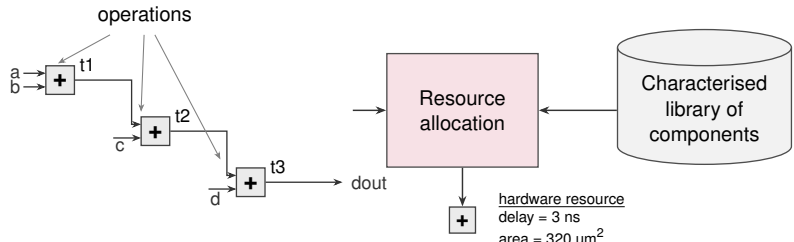
Each operation becomes a node; each variable an edge between two operations.

This three-adder graph, drawn the same way, comes back until the end of Part 4.

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Step 2: resource allocation

Which operators, how wide, how fast, how big

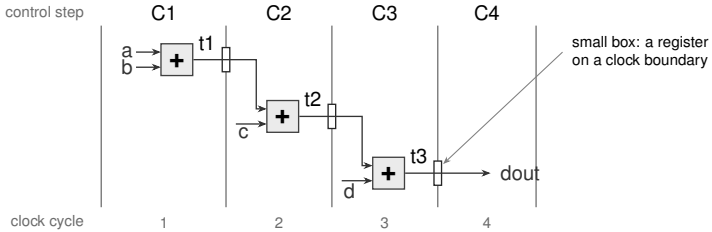


- Choose the operators and estimate the datapath widths: `int + int` gives a 32-bit adder
- Estimate the area and the delay of each operator, from the technology data of the target (on an FPGA: logic and DSP blocks)

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Step 3: scheduling

Each operation is placed in a clock cycle



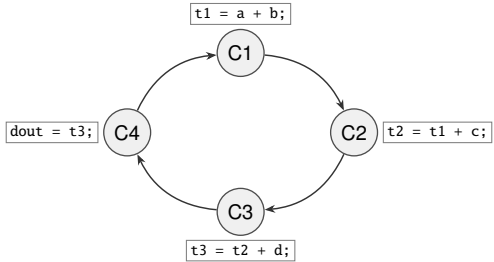
- Assumption: $T_{addition} < T_{clk}$: one addition per clock cycle
- Each operation is assigned to a clock cycle; intermediate results are stored in registers

The clock decides. A slower clock lets two additions *chain* in one cycle. Intel HLS: `hls_scheduler_target_fmax_mhz(50)`.

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Step 4: control

A finite state machine drives the chosen schedule

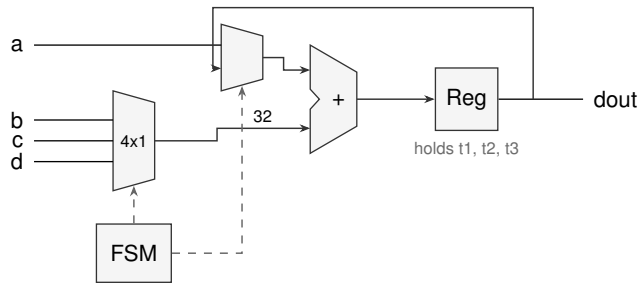


- A finite state machine, associated with the data path, drives the chosen schedule: one state per control step
- C4 writes the result; the next call can then start in C1

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

“Hardware implementation”

A constraint-free design: binding operations to hardware



- **Binding**: one adder does all three additions; one register holds t1, then t2, then t3
- Minimum resources: one adder, a multiplexer on each adder input, a register and an FSM. Sharing is not free: the multiplexers and the FSM are the price

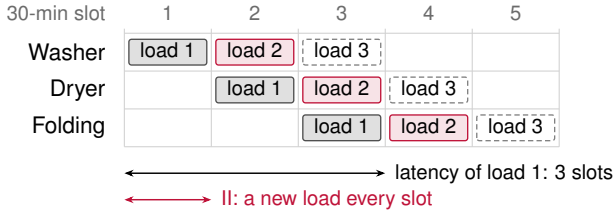
Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Outline

Introduction: what HLS is and how it works
Principles: one example in four steps
Latency, II and pipelining
Loop unrolling
A real loop, and the hints that steer the tool
Limits and choices
Interfaces and practice

Two numbers describe any hardware block

The laundry picture



- **Latency (L)**: how long one load takes, from washer to folded
- **Initiation interval (II)**: how often a new load can start

Each load still takes 1.5 hours, yet a load comes out every 30 minutes: throughput is set by the II, not by the latency.

Pipelining successive calls

Back to accumulate. No constraint: L = 3, II = 4

accumulate is called again and again on new inputs. Pipelining overlaps successive **calls**: their II is the **component II** in Intel HLS.

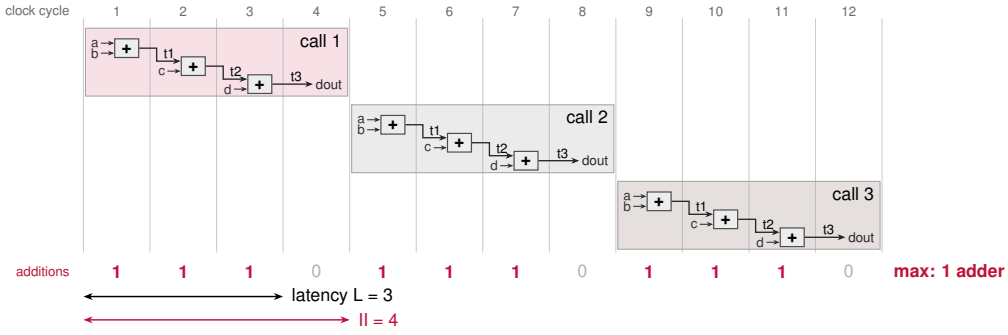
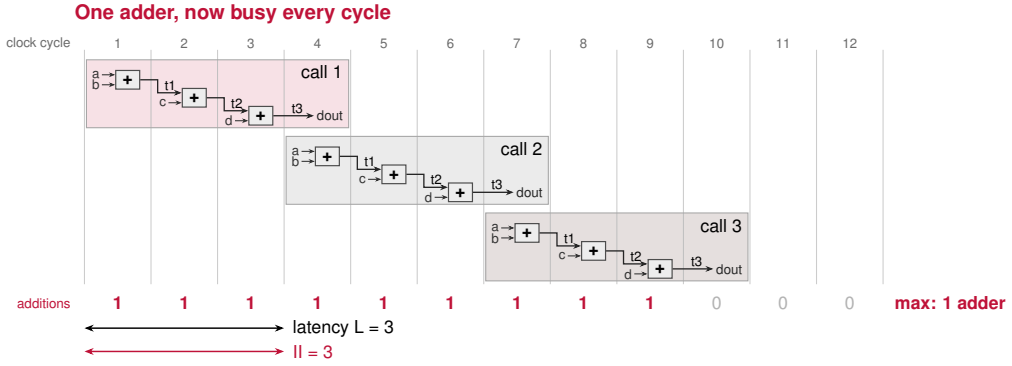


Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Pipelining successive calls: II = 3, L = 3

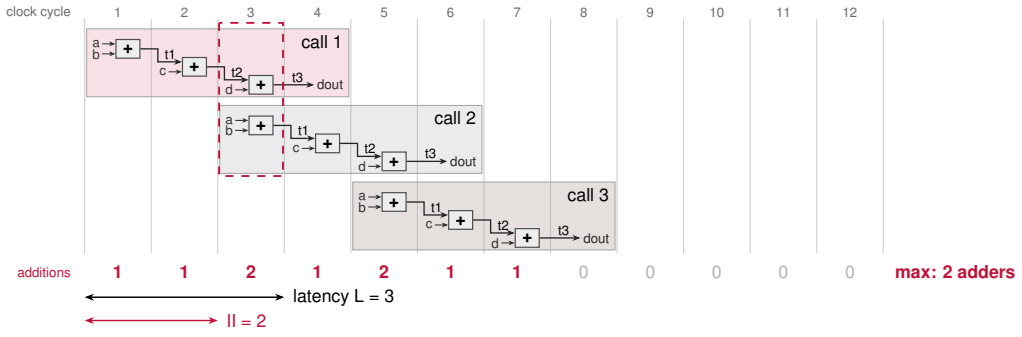


- In cycle 4, call 1's output is written while call 2's inputs are read; the adder never waits
- Requested with a pragma or a script. Intel HLS: `hls_component_ii(3)` before the component

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Pipelining successive calls: II = 2, L = 3

Two additions in the same cycle: two adders



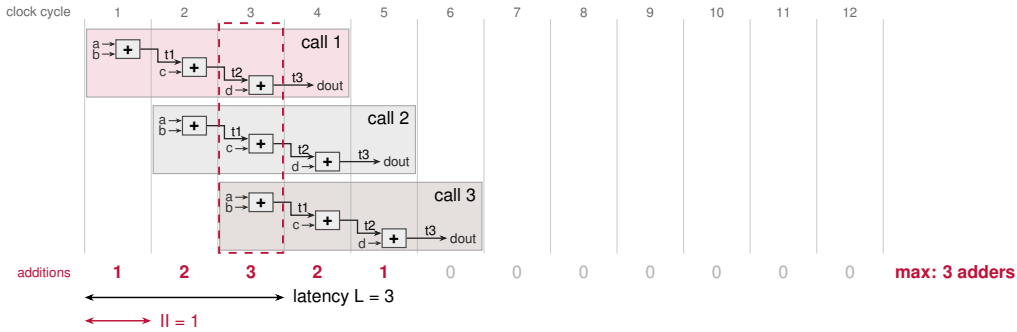
- In cycle 3 (red box), call 1 computes its last addition while call 2 computes its first

Ask the room. How many adders for II = 1?

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Pipelining successive calls: II = 1, L = 3

Three adders in parallel



Latency never moved: L = 3 at every II. Only how often a new call starts changed, and with it the number of adders.

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Outline

Introduction: what HLS is and how it works
Principles: one example in four steps
Latency, II and pipelining
Loop unrolling
A real loop, and the hints that steer the tool
Limits and choices
Interfaces and practice

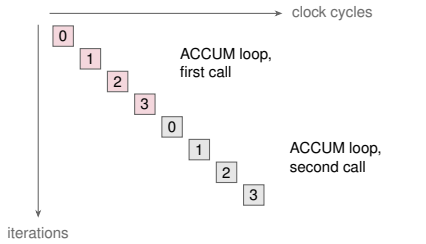
Loop unrolling

Parallelism inside one call

- Successive calls do not overlap
- Instead, we add parallelism inside the loop

```
void accumulate4(int din[4], int &dout)
{
  int acc = 0;
  ACCUM: for (int i = 0; i < 4; i++)
    acc += din[i];
  dout = acc;
}
```

Example after the Blue Book: `din` is 4×32 bits

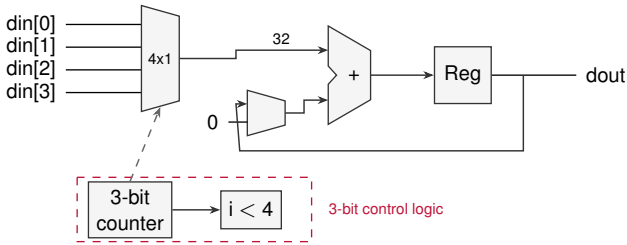


Initial schedule: two successive calls, one iteration per clock cycle

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Loop unrolling: initial implementation, II = 4

The loop as written

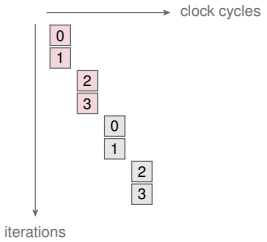


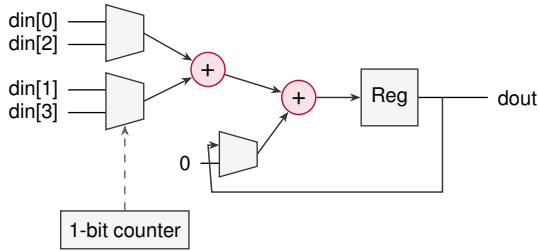
- First cycle: `din[0] + 0`; the other three cycles accumulate
- A 3-bit counter controls the loop: 1 adder, II = 4. No pragma: this is the code as written

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Loop unrolling: partial unrolling by 2, II = 2

```
#pragma unroll 2
```





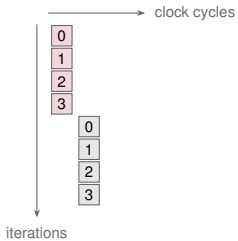
two iterations per clock

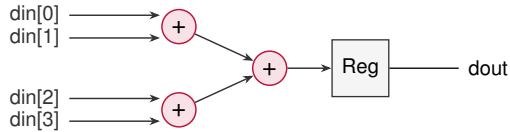
- Two inputs read in the same clock cycle
- Two adders (one adds the pair, one accumulates) and a 1-bit counter

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Loop unrolling: full unrolling, II = 1

```
#pragma unroll
```





four iterations per clock

- Four inputs read in the same clock cycle
- Three adders (an adder tree), no counter: the loop has disappeared

Figure redrawn after M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics.

Pipelining versus unrolling

	Pipelining	Unrolling
Parallelism	global: successive calls (or loop iterations) overlap	local: inside one call (or one iteration)
A lower II	first uses idle hardware (II 4 $\rightarrow$ 3, same adder), then needs more operators	more operators, by construction
In a given cycle	inputs and outputs belong to several calls	inputs and outputs belong to one call

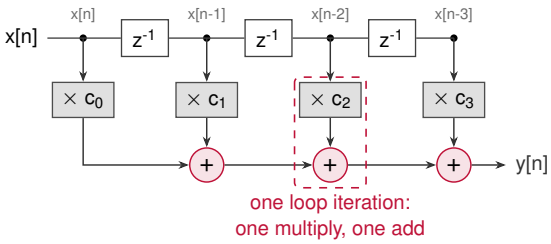
Pipelining can first use idle hardware; only a more aggressive II needs more operators. Unrolling creates parallel operations explicitly.

Outline

Introduction: what HLS is and how it works
Principles: one example in four steps
Latency, II and pipelining
Loop unrolling
A real loop, and the hints that steer the tool
Limits and choices
Interfaces and practice

Meet the FIR filter

A weighted sliding window: the loop at the heart of Lab 2




```
// one output sample
acc = 0;
for (k = 0; k < 4; k++)
  acc += c[k] * x[k];
y = acc;
```

z^{-1}: the previous sample. Each iteration: one multiply and one add.

Lab 2. The same loop with 256 taps, removing an 8 kHz tone from a piano recording on the DE1-SoC.

FIR: the plain loop, then pipelined

Same hardware, no idle slot

Plain loop: 8 cycles

cycle	1	2	3	4	5	6	7	8
Multiplier	×0		×1		×2		×3	
Adder		+0		+1		+2		+3

Pipelined loop: 5 cycles

cycle	1	2	3	4	5	6	7	8
Multiplier	×0	×1	×2	×3				
Adder		+0	+1	+2	+3			

- Both: **1 multiplier, 1 adder**; in the plain loop each sits idle half the time
- Pipelined: the multiplier starts the next tap while the adder ends the previous one. Cost: registers

The tool already does this. Intel HLS pipelines loops by default.
`#pragma disable_loop_pipelining` gives the plain loop.

FIR, version 3: unrolling by 2

`#pragma unroll 2`, as in Lab 2 STEP6

cycle	1	2	3	4	5	6	7	8
Multiplier 1	×0	×2						
Multiplier 2	×1	×3						
Adder 1: pair		p0+p1	p2+p3					
Adder 2: total			acc +	acc +				

Two taps per iteration: the multipliers work in pairs, adder 1 adds each pair of products, adder 2 adds the pair to the running total. **2 multipliers, 2 adders, 4 cycles** (instead of 5).

> **What it costs.** Real operators: on an FPGA every extra multiplier is DSP blocks. Unrolling copies the loop body; full unrolling (`#pragma unroll`) copies it four times and adds the products with a tree.

The FIR versions, and one rule for both knobs

Same computation: different hardware, latency and output rate

Version	Multipliers	Adders	Latency	Interval
Plain loop	1	1	8	8
Pipelined loop	1	1	5	5
Unrolled $\times 2$, pipelined	2	2	4	4
Fully unrolled	4	3	3	3
Fully unrolled, calls overlapped	4	3	3	1

Illustrative, one cycle per operation. Latency: cycles for one output. Interval: cycles between two outputs.

A given II needs enough operators to do one iteration's work every II cycles.

- accumulate: 3 additions per call, so II = 1 needs 3 adders, even without unrolling
- The FIR loop: 1 multiply and 1 add per iteration, so loop II = 1 needs 1 of each; unroll by 2 and it needs 2 of each

Where the hints go

Four places in the code (Intel HLS)

```
#include "HLS/hls.h"

hls_scheduler_target_fmax_mhz(50)              // 1
component void fir4(
  ihc::stream_in<int>  &x,                     // 2
  ihc::stream_out<int> &y,                     // 2
  hls_avalon_agent_register_argument int gain) // 2
{
  hls_register static int win[4];              // 3
  const int c[4] = {1, 3, 3, 1};
  #pragma unroll
  for (int k = 3; k > 0; k--) win[k] = win[k-1];
  win[0] = x.read();           // one sample per call
  int acc = 0;
  #pragma unroll                               // 4
  for (int k = 0; k < 4; k++)
    acc += c[k] * win[k];
  y.write(gain * acc);
}
```

1. **Before the component**: how the whole block is scheduled
2. **On an argument**: port and protocol
3. **On a variable**: registers or RAM
4. **Before a loop**: unroll, II, pipelining

A hint is a request. Unrolling pays here because the window is in registers. Unroll a loop around `x.read()` and the stream still gives one sample per cycle.

The hints you will use in the labs

Intel HLS Compiler 21.4

Hint	Where	Effect	Lab
`component`	before the function	becomes hardware; what it calls is inlined	1–3
`ac_int<W,S>`	a type	exact bit width W, signed or not	1, 2
`hls_scheduler_target_fmax_mhz(N)`	before the component	clock the scheduler aims for (default 240 MHz)	2
`#pragma unroll N`, `#pragma unroll`	before a loop	copy the loop body N times, or completely	2, 3
`ihc::stream_in<T,...>`, `ihc::stream_out<T,...>`	argument type	Avalon-ST stream; options such as `ihc::bitsPerSymbol<8>`	2
`hls_avalon_agent_register_argument`	before an argument	a register written by the CPU	2, 3
`hls_avalon_agent_component`	before the component	started and polled through CPU registers	3
`ihc::mm_host<T,...>`	argument type	memory port: `aspace`, `awidth`, `dwidth`, `latency`, `waitrequest`	3
`ihc_hls_enqueue`, `ihc_hls_component_run_all`	testbench	queue calls, run them back to back	1, 2

Outline

Introduction: what HLS is and how it works
Principles: one example in four steps
Latency, II and pipelining
Loop unrolling
A real loop, and the hints that steer the tool
Limits and choices
Interfaces and practice

Four questions before you turn a knob

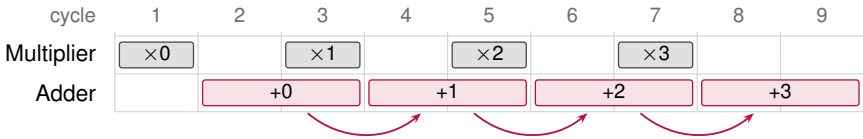
Pipelining and unrolling only help when the data allows it

1. **Does the next step need my result?**
   Then it has to wait for it, however much hardware you add.
2. **Is the data already here?**
   You cannot compute faster than the data arrives.
3. **Is the memory port free?**
   A memory serves only one or two accesses per cycle.
4. **Do I know how many iterations there are?**
   A loop whose length depends on the data cannot be fully unrolled.

When the tool does not reach the II you hoped for, the reason is almost always one of these four. The Loop Analysis report tells you which.

1. Waiting on yourself: the running total

A loop-carried dependency



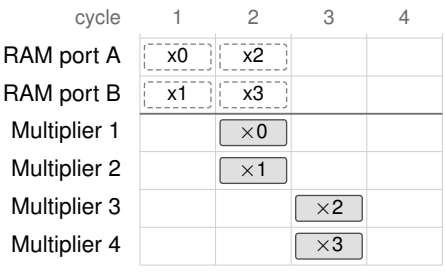
Same pipelined FIR loop, but one addition now takes 2 cycles, and each `acc +=` needs the previous `acc`: **one tap every 2 cycles, 9 cycles instead of 5**. More hardware would not help.

The clock changes the answer. At a demanding clock target the addition may take several cycles. Since each accumulation needs the previous result, the requested II may not be reached.

Lab 2, STEP3. At the default target (about 240 MHz) the accumulator limits the loop II; at 50 MHz it does not. Predict both, with loop cycles $\approx (N - 1) \times II + L$.

3. Is the memory port free?

A RAM serves one or two accesses per cycle



Unroll the FIR by 4. In this example the samples sit in one on-chip RAM with two ports: two reads per cycle.

You paid for 4 multipliers and get the speed of 2.

> **Ways out.** Split the array over several memories (`hls_numbanks(4)`) or keep it in registers (`hls_register`): cheap for 4 samples, not for 256.

Every design has a deadline

Faster is not better

Lab 2 in one line

- The board clock runs at 50 MHz
- The audio codec delivers one stereo frame at 48 kHz
- So the filter has about **1041 cycles per frame**

A filter that finishes in 300 cycles then waits 741 cycles for the next frame. Those extra multipliers did nothing.

The best design is the smallest one that meets its deadline.

Watch out. The deadline comes from the *system* (the codec, the bus, the CPU), not from the tool. Find it before you optimise anything.

Pick one

The 4-tap FIR again

Version	Multipliers	Adders	Cycles between outputs
Pipelined loop	1	1	5
Unrolled $\times 2$	2	2	4
Fully unrolled, block pipelined	4	3	1

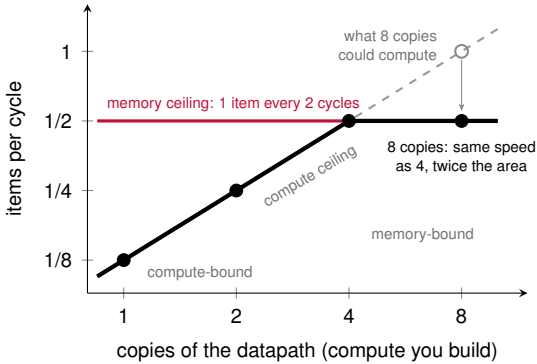
A new sample every 6 cycles. Which version do you build?

Every 4 cycles. And now?

Every cycle. And now?

Two ceilings: compute and memory

A simple roofline: more compute only helps up to the memory ceiling



- Memory: each item needs one read and one write through one port, so at most 1 item every 2 cycles
- Compute: one copy of the datapath does 1 item every 8 cycles; unrolling adds copies
- Throughput is the **lower** of the two ceilings

Below the roof, more compute helps. Above it, it only costs area.

Lab 3. Draw it for your DES before you synthesise.

Design space exploration

In one picture

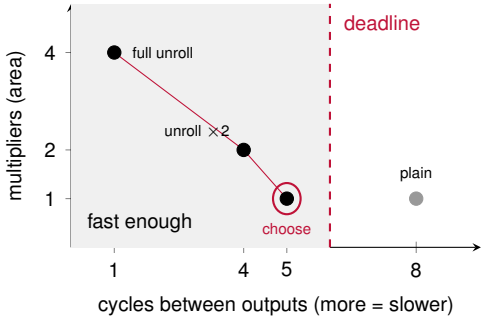


Illustration. Red: Pareto front

- Several architectures of one C++ function: unrolling, II, clock target, widths, memories
- Keep correctness first; then compare fitted area with component II
- Pareto front: the variants that no other variant beats on both axes
- Choose the cheapest one that meets the constraint; then look for cheaper still: share operators, narrow the widths

Lab 2: 50 MHz / 48 kHz $\approx$ 1041 cycles per stereo frame. Faster than that is not better.

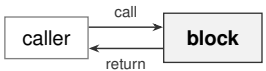
Outline

Introduction: what HLS is and how it works
Principles: one example in four steps
Latency, II and pipelining
Loop unrolling
A real loop, and the hints that steer the tool
Limits and choices
Interfaces and practice

C/C++: communication ports

The hardware is a function: its inputs and outputs become ports

Call and return



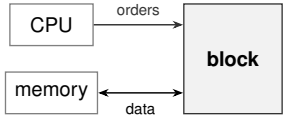
a waiter takes the order
and brings the dish

Streams



a conveyor belt:
items move one by one

Registers + memory



a mailbox for orders;
the block fetches its own data

- The designer must choose the protocol of each port: Vitis HLS with pragmas, Intel HLS with types and attributes
- The same algorithm can be wrapped in any of the three; the choice decides how fast data really gets in and out

Interfaces in Intel HLS

Types and attributes instead of pragmas

Interface	In the C++ code	Lab
Call and return handshake	default (start and busy, done and stall)	1
Avalon-ST stream	`ihc::stream_in<T>`, `ihc::stream_out<T>`	2
Register written by a CPU	`hls_avalon_agent_register_argument`	2, 3
Started and polled through registers	`hls_avalon_agent_component`	3
Access to system memory	`ihc::mm_host<T, ...>`	3

Arguments passed by value become input ports; pointers and references become Avalon-MM host ports, unless a type or attribute says otherwise.

Older names. Since 21.2 Intel says *agent* and *host*; `hls_avalon_slave_register_argument` and `ihc::mm_master` still compile but are deprecated.

The testbench and the flow

Four checkpoints, each slower and closer to the truth

```
int main() {
  unsigned int result[SIZE];
  for (int i = 0; i < SIZE; i++)
    ihc_hls_enqueue(&result[i],
                    &simple_counter);
  ihc_hls_component_run_all(
                    simple_counter);
  // check result[] = 0, 1, 2...
}
```

- `main()` is the testbench: it calls the component and checks the results
- Queued calls run back to back, so the co-simulation measures the real II

Step	What it answers	Time
g++ with the HLS headers	Is the C++ correct?	seconds
`i++ -march=<FPGA part>`	Which hardware? (RTL and reports)	minutes
Run the i++ executable	Does the RTL behave like the C++?	minutes
Quartus compile	Real area and Fmax	the longest

How every lab works

1. Predict

Before each run, write down the II, the number of multipliers and the number of memories you expect.

2. Measure

Run the flow and read the reports and the waveforms.

3. Explain

Explain every difference between prediction and measurement, using the reports.

- **Correctness first.** Each lab has a self-checking testbench and a golden reference. A smaller design that gives wrong results does not count.
- **Choose with numbers.** Labs 2 and 3 end with a results table and a justified choice.

What to remember

1. HLS: you describe *what*; the tool decides *how*: graph, allocation, scheduling, binding and control
2. Hardware trades space for time. Every operator and every bit costs area
3. Latency is how long one call takes; II is how often a new one can start
4. Pipelining overlaps calls; unrolling works in parallel inside one. A given II needs enough operators for one iteration's work every II cycles
5. Hints (pragmas, attributes, types) are requests: check the reports
6. The data decides what is possible: dependencies, arrival, memory ports, loop counts
7. The best design is the smallest one that meets its deadline; compute above the memory ceiling is wasted. Predict, measure, explain

Given a C++ function and a constraint, which hardware should we build, and how do we know?

Going further

Not presented in the lecture: for reading, and for the labs

HLS in context

Methodologies, the FFT example, when HLS pays off, every bit costs

Pipelining, unrolling and hints

The FIR versions one by one, other Intel HLS hints, the same hints in Vitis HLS

When the data decides

Breaking a dependency, data arrival, loop counts, shrinking a design

Interfaces

A stream transfer (valid and ready), SystemC ports, data types for DSP, memory-bound designs

Practice

HLS tools today, the FPGA case, an Intel HLS component, reading the reports, today's ideas mapped to the labs, references

HLS methodologies and tools

A plethora of situations

Input language

C, C++ or SystemC

Two audiences

Computer scientists know high-level languages, not hardware architectures.

Electronics engineers know hardware, not high-level languages.

Implicit target architecture

CPU style: a control flow graph and predefined resources (register bank, ALUs, multipliers). Hardly used any more.

Dataflow style: a data flow graph and state machines. Most tools today, including the one in the labs.

HLS methodologies and tools

A “long” learning time

- A paradigm shift: from describing hardware (RTL) to describing an algorithm and constraints (HLS)
- Moving from one HLS tool to another is not easy
- Tool tutorials use very simple examples; but once mastered, development is faster

Example: a beginner codes a FIR filter (Lab 2)

RTL	Easy, if you know in advance the parallelism and pipelining you want
HLS	You steer the tool with pragmas to get them, and the interface timing it generates may not fit your system

What question do we want to solve?

Example: an FFT

Optimised FFT IP cores exist. With HLS we start from a reference C code:

1. Rewrite
the C code in the subset the tool accepts

→

2. Choose pragmas
unroll loops or not; pipeline; data in single-port RAM, dual-port RAM or registers

→

3. Explore
compare the combinations, with scripts or tool support

Step 3 is slow: every point needs a synthesis. Lab 2 (STEP6) is this exploration, with measurements.

When HLS pays off, and when it doesn't

HLS shines for

- large datapaths: filters, transforms, ciphers
- standard interfaces (streams, buses, CPU registers) that would take days to write by hand
- comparing several architectures quickly

HLS is the wrong tool for

- tiny, control-only designs: a counter, a small state machine
- cycle-exact protocols you must control yourself

Lab 1. A C++ counter goes through the whole flow and is compared with hand-written Verilog. It works, but adds a handshake you never asked for: know when *not* to use HLS.

Every bit costs

Widths are area, and behaviour

```
unsigned int     cnt;   // 32 flip-flops
ac_int<8,false> cnt8;   //  8 flip-flops
```

- In software you pick `int` and move on; in hardware, width is area
- Any width with the Algorithmic C types: `ac_int<W,S>`, fixed point `ac_fixed<W,I,S>`; Intel adds `hls_float`

Width is also behaviour. An 8-bit counter wraps after 255. Choosing a width is a functional decision.

Multipliers are the expensive part. FPGAs have fixed-size multiplier blocks (DSP blocks). On Cyclone V the largest is 27×27 bits: a 32×24-bit product needs more than one.

In the lab. Lab 1 tries an 8-bit counter, then 300 calls. Lab 2 uses 32-bit coefficients: do they need to be that wide?

Same latency, four throughputs

What pipelining accumulate costs

II (cycles between calls)	Latency L	Adders	What changed
4	3	1	nothing overlaps
3	3	1	output written while the next inputs are read
2	3	2	two additions in the same cycle
1	3	3	every addition runs every cycle

Latency never moved: each call still takes 3 cycles. Only how often a new call starts changed.

Lab 1. Streaming (audio, video) cares about II; a single request-and-wait cares about latency. Read both in the reports, then measure them on the waveforms.

Loops: the core of the HLS tools

In RTL, by hand

- coding the state machines, including the start and the end of a pipeline
- debugging them
- usually, only one alternative ever gets coded

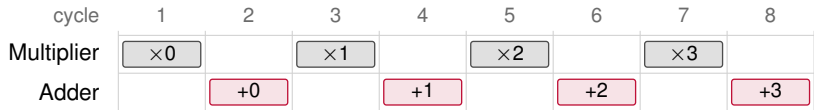
In HLS, automatic

- for simple loops with a known number of iterations
- pipelining and unrolling, alone or combined
- so exploring alternatives is cheap

Warning: the best result can be counterintuitive. All tools do this, but not with the same vocabulary.

FIR, version 1: the plain loop

One iteration finishes before the next one starts



1 multiplier, 1 adder, 8 cycles per output.

Note. Look at the empty slots: the multiplier sits idle half the time, and so does the adder.

Ask the room. Without adding any hardware, can we use those empty slots?

FIR, version 2: pipelining the loop

Same hardware, no idle slot

cycle	1	2	3	4	5	6	7	8
Multiplier	×0	×1	×2	×3				
Adder		+0	+1	+2	+3			

The multiplier starts the next tap while the adder finishes the previous one. **Same hardware, 5 cycles.**

What it costs. A few registers between the stages. Registers are cheap.

The tool already does this. Intel HLS pipelines loops by default (here at loop II = 1); Vitis HLS does it automatically for long loops. `#pragma ii N` requests another loop II; `#pragma disable_loop_pipelining` turns it off.

Other hints worth knowing

Not needed in the labs, useful in your projects

Applies to	Hint	Effect
Loop	`#pragma ii N`	request a loop II of N
	`#pragma disable_loop_pipelining`	one iteration at a time: less area, less throughput
	`#pragma max_concurrency N`	at most N iterations in flight: fewer memory copies
	`#pragma loop_coalesce N`	merge N nested loops into one
	`#pragma ivdep`	ignore memory dependencies between iterations
Component	`hls_component_ii(N)`	request a component II of N
	`hls_max_concurrency(N)`	at most N calls in flight
Variable	`hls_register`, `hls_memory`	keep it in registers, or in on-chip RAM
	`hls_numbanks(N)`, `hls_bankwidth(N)`	split a memory into banks: more accesses/cycle
	`hls_singlepump`, `hls_doublepump`	memory clocked at the component rate, or twice it

`ivdep`. If the dependency is real the hardware is wrong, and g++ ignores the pragma: only co-simulation catches it.

The same hints in Vitis HLS

Same ideas, different spelling

Intel HLS	Vitis HLS
`#pragma unroll N`	`#pragma HLS UNROLL factor=N`
`#pragma ii N`	`#pragma HLS PIPELINE II=N`
`hls_scheduler_target_fmax_mhz(N)`	clock period set in the project or script
`hls_numbanks`, `hls_register`	`#pragma HLS ARRAY_PARTITION`
`ihc::stream_in<T>`, `ihc::stream_out<T>`	`#pragma HLS INTERFACE axis`
`hls_avalon_agent_register_argument`	`#pragma HLS INTERFACE s_axilite`
`ihc::mm_host<T, ...>`	`#pragma HLS INTERFACE m_axi`
`ac_int<W,S>`	`ap_int<W>`, `ap_uint<W>`

```
void example(int A[50], int B[50]) {
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B
  for (int i = 0; i < 50; i++) {
#pragma HLS PIPELINE II=1
    B[i] = A[i] + 5;
  }
}
```

AMD Vitis HLS: two AXI4-Stream ports and a pipelined loop.

Intel HLS picks a port protocol with a *type*, Vitis HLS with a *pragma*: same ideas.

Breaking the chain

Going further: a code change, not a pragma

One running total

Every addition waits for the previous one. 256 taps, 2 cycles per addition: about 512 cycles.

Two running totals

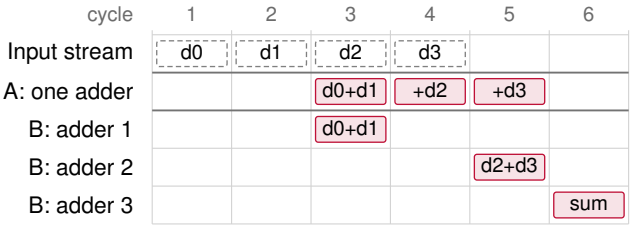
Even taps into one, odd taps into the other, add the two at the end. Each total only waits for itself, so a new tap can start every cycle: about 256 cycles, plus the final addition.

Going further. You trade one extra adder and register for half the time.

Floating point. With integers the result is identical. With `float` the order of additions changes the rounding, so tools will not do it for you unless you allow it.

2. Is the data already here?

You cannot compute faster than the data arrives



Four numbers arrive on a stream, one per cycle.

A (one adder) adds them as they come: done in cycle 5.
B (unrolled tree, three adders) must wait for d3: done in cycle 6.

> **Note.** Unrolling only helps when the inputs are all available at once. In Lab 2, audio arrives once every 1041 cycles: the input sets the pace.

4. Do I know how many iterations there are?

Fixed count

```
for (k = 0; k < 256; k++)
  acc += c[k] * x[k];
```

The tool knows the loop runs 256 times: it can unroll it fully and predict its timing.

Count depends on the data

```
while (x != 0) {
  x = step(x);
}
```

Nobody knows how many copies to build: no full unrolling, and the timing depends on the input.

> **A common fix.** Loop up to a known maximum and decide inside the loop whether this iteration does anything. The hardware is sized for the worst case.

Knobs that shrink, not just speed up

When you are already fast enough

Share

Left and right channels can use the same multiplier, one after the other, if the deadline allows.

Exploit symmetry

Many filters have mirror-image coefficients ($c_0 = c_3$, $c_1 = c_2$). Add the two samples first, multiply once: half the multipliers.

Narrow the widths

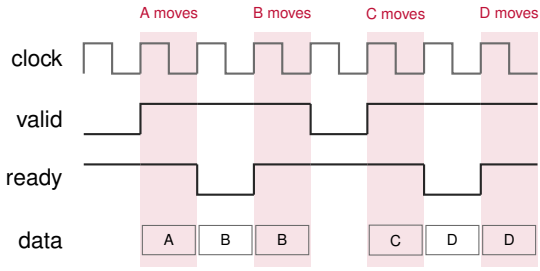
Fewer bits per coefficient can fit in one DSP block. Check that the output error stays acceptable.

Estimates vs. reality. HLS reports give estimates; Quartus gives the real area and Fmax.

Lab 2, STEP6. Your design space exploration: a results table, one plot, a justified choice.

A stream transfer

Data moves only when both sides agree



- **valid**: the sender has an item
- **ready**: the receiver can take it
- An item moves only on a cycle where *both* are high (shaded)
- Otherwise the sender holds the same item (B, then D)

Measuring II on waveforms. Count only the cycles where valid and ready are both high. The distance between two such cycles is the II you actually achieved.

SystemC: communication ports

Explicit hardware constructs

- The tools recognise SystemC constructs such as `sc_in`, `sc_out` or `sc_fifo`
- Ports and processes are explicit: closer to RTL, further from plain C++

Today. Vivado HLS accepted SystemC; its successor Vitis HLS no longer does. Catapult and Stratus still do.

```
SC_MODULE(sc_sequ_cthread) {
  sc_fifo_out<int> dout;
  sc_fifo_in<int>  din;
  ...
};
```

Mathematical support

Data types for DSP

DSP needs control over the precision of every value. What each language offers:

	Any-width integers	Fixed point	Floating point
VHDL / Verilog	yes	limited	limited
SystemC	yes	yes	`float`, `double`
C++	standard sizes only	no	`float`, `double`
C++ with AC datatypes	`ac_int<W,S>`	`ac_fixed<W,I,S>`	`ac_float`

Algorithmic C (AC) datatypes: templated C++ classes. Intel HLS provides `ac_int`, `ac_fixed` and its own `hls_float`.

Going further: when memory is the limit

Memory-bound versus compute-bound

A block that fetches its own data from memory, through one port:

- each 64-bit block of data needs one read and one write
- one port does one access per cycle
- so at best one block every 2 cycles, **however fast the computation is**

A cipher able to take a new block every cycle is then wasted hardware.

Going further. Speed is limited by whichever is slower: the computation, or how fast data moves in and out. Find which one before adding compute.

Lab 3. How much DES hardware is worth building when the data comes from memory?

Some known HLS tools

Tool	Vendor	Input language	Target
Stratus HLS	Cadence	C++, SystemC	ASIC, FPGA
Catapult	Siemens EDA (formerly Mentor)	C++, SystemC	ASIC, FPGA
Vitis HLS	AMD (formerly Xilinx)	C, C++	AMD FPGAs
HLS Compiler	Intel (FPGAs now Altera), discontinued	C++	Altera FPGAs
SmartHLS	Microchip (formerly LegUp)	C++	Microchip FPGAs
Bambu	Politecnico di Milano (open source)	C, C++	FPGA, ASIC

The landscape keeps moving. The Intel HLS Compiler used in the labs (21.4) is discontinued. FPGA support in Intel oneAPI, its proposed successor, was itself removed in oneAPI 2025.1. Altera, independent since September 2025 (Silver Lake 51%, Intel 49%), now supplies its own FPGA tools. The concepts are the same in every tool.

The FPGA case

Heterogeneous architectures

OpenCL

- parallel programming, first for CPUs and GPUs
- SPMD: single program, multiple data
- extended to CPU + FPGA systems

Platforms

- SoC FPGAs: CPU and FPGA in the same chip (the Cyclone V SoC of the DE1-SoC board)
- FPGA accelerators in data centres

In general: one tool for each target platform.

An Intel HLS component

```
#include "HLS/hls.h"

component unsigned int
simple_counter() {
  static unsigned int cnt = 0;
  return cnt++;
}
```

Lab 1

- `component`: the function that becomes hardware; the functions it calls are inlined
- Clock and reset are implicit
- A `static` variable keeps its value between calls: a register, reset to its initial value
- Default interface: a call handshake (start, busy) and a return handshake (done, stall)
- The return type sets the output width (32 bits here)

Reading the reports

Throughput

Loop Analysis: is each loop pipelined, with which II, and what limits it (for example a loop-carried dependency)

Verification Statistics: latency and II measured in co-simulation

Area

Area Analysis: HLS estimates for logic, registers, memory and DSP blocks

Quartus summary: fitted area and Fmax under the project's SDC constraint

The HLS target is a scheduling request; the estimated Fmax and Quartus fitted Fmax are separate. A standalone fit is not board-system timing.

From today's ideas to the labs

Idea from today	Where you meet it	What you measure
The four steps; latency and II; every bit costs	Lab 1: counter	II and latency on waveforms; area
Pipelining and unrolling; `#pragma unroll`	Lab 2: audio FIR	loop II, component II, multipliers
Waiting on yourself; the clock target	Lab 2, STEP3	loop II at 240 vs. 50 MHz
Enough is enough (1041 cycles)	Lab 2, STEP6	smallest design that keeps up
Streams; agent registers	Lab 2	valid/ready transfers; bypass register
Memory host; memory-bound	Lab 3: DES	block throughput vs. area

References

- These slides are based on the original ICS901 HLS lecture by Yves Mathieu (Télécom Paris)
- M. Fingeroff, *High-Level Synthesis Blue Book*, Mentor Graphics, 2010 (the accumulate example and the figures of Parts 2 to 4, redrawn)
- Intel High Level Synthesis Compiler Pro Edition: Reference Manual and User Guide, version 21.4 (the version installed for the labs): loop pragmas, component and memory attributes, interfaces
- Algorithmic C datatypes: `github.com/hlslibs/ac_types`
- AMD, *Vitis HLS User Guide* (UG1399)
- S. Williams, A. Waterman, D. Patterson, “Roofline: an insightful visual performance model for multicore architectures”, *Communications of the ACM*, 2009