Reconfigurable Architecture (9)

High-level synthesis (1 of 2)

Today

- HLS: High-Level Synthesis technology
  - Generate RTL from C, C++, Java, System C or other languages
  - OpenCL (mostly used in GPUs) is also supported

HLS tools

- Without particular targets
  - Impulse C (Impulse), Cyber WorkBench (NEC)
- For specific device technology
  - Vivado HLS (Xilinx)
- For specific platform
  - Carte (SRC: C/Fortran), MaxCompiler (Maxeler: Java)

Better in HLS:

- Implementing complex algorithms: hard to implement with HDLs
  - Describe and verify as a software
    - Debug without HDL simulator but with compilers: very fast
  - Easy to modify / update algorithm
- Easy to try design trade-offs: i.e, area vs speed

Popular HLS targets

- Financial applications: stock and FX predictions
- Signal processing: Digital filter implementations in HLS
  - Also effective for image processing, especially real-time ones
  - Matlab is also a popular design entry

HLS is not a magic bullet

- Not always fast, may result in a (very) large circuit
  - Tuning is crucial for HLS designs
- Not good at clock-accurate stuff as HDL
  - HDL is better for external interface logic
  - HLS is suitable for internal, complicated algorithms

How HLS tools work

- Extract control flow
  - Branches and loops
- Then, extract data flow
  - For each BBs (basic blocks)
  - Perform schedule and bind

```
void foo(int in[3],
         char a, char b, char c,
         int out[3]) {

  int x, y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

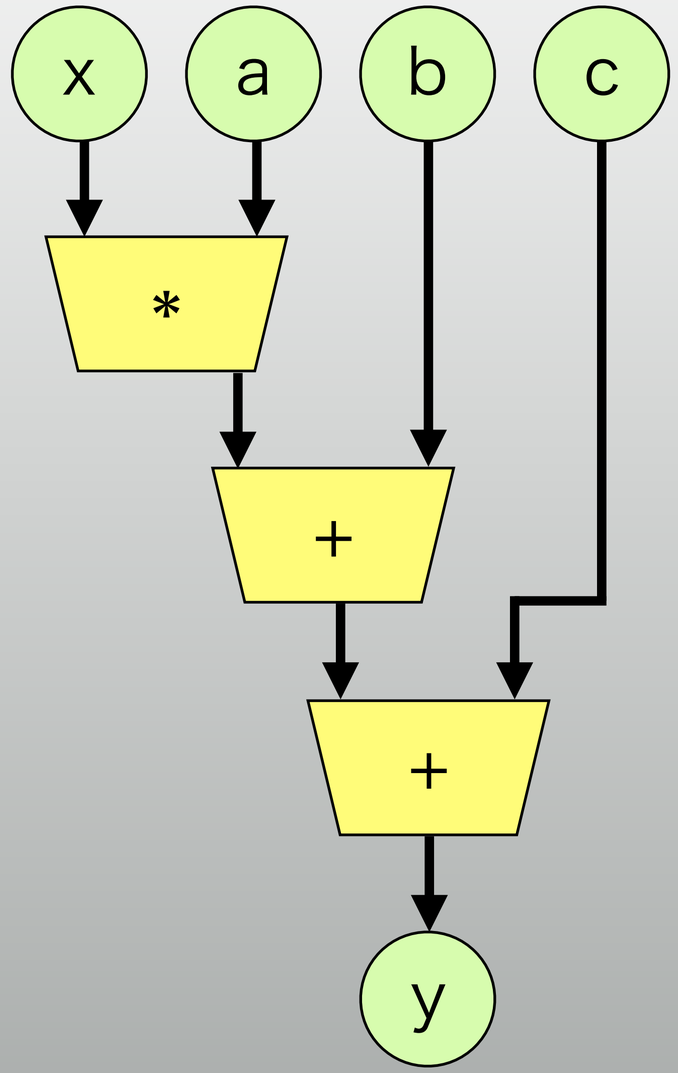
Basic Block

Loop

Xilinx UG902 (v2014.1)

Data flow extraction

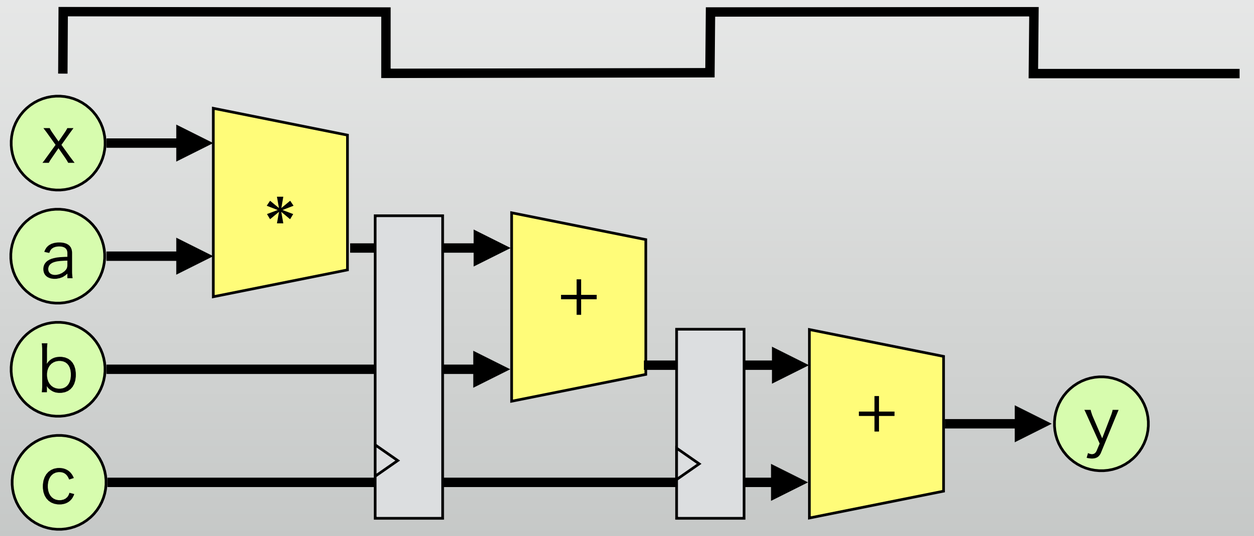
```
int foo(char x, char a,
        char b, char c){
  char y;
  y = x*a+b+c;
  return y;
}
```



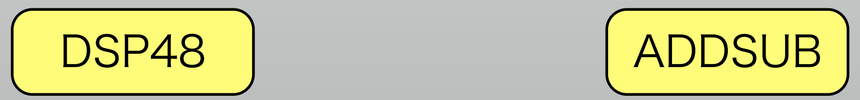
Xilinx UG902 (v2014.1)

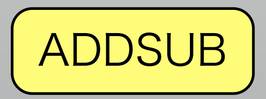
Scheduling and binding

Scheduling

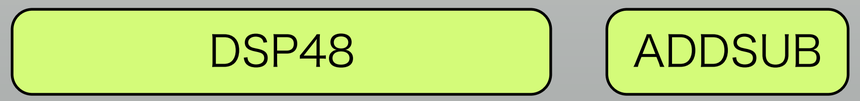


Initial Binding





Target Binding



Interface synthesis

- Scalar arguments:
  synthesized as I/O ports
- Array arguments: basically
  synthesized as BlockRAM I/Fs
  - Communication via RAMs
- Details later

```
void foo( int in[3],
          char a, char b, char c,
          int out[3]) {

  int x, y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

Xilinx UG902 (v2014.1)

Loop optimizations

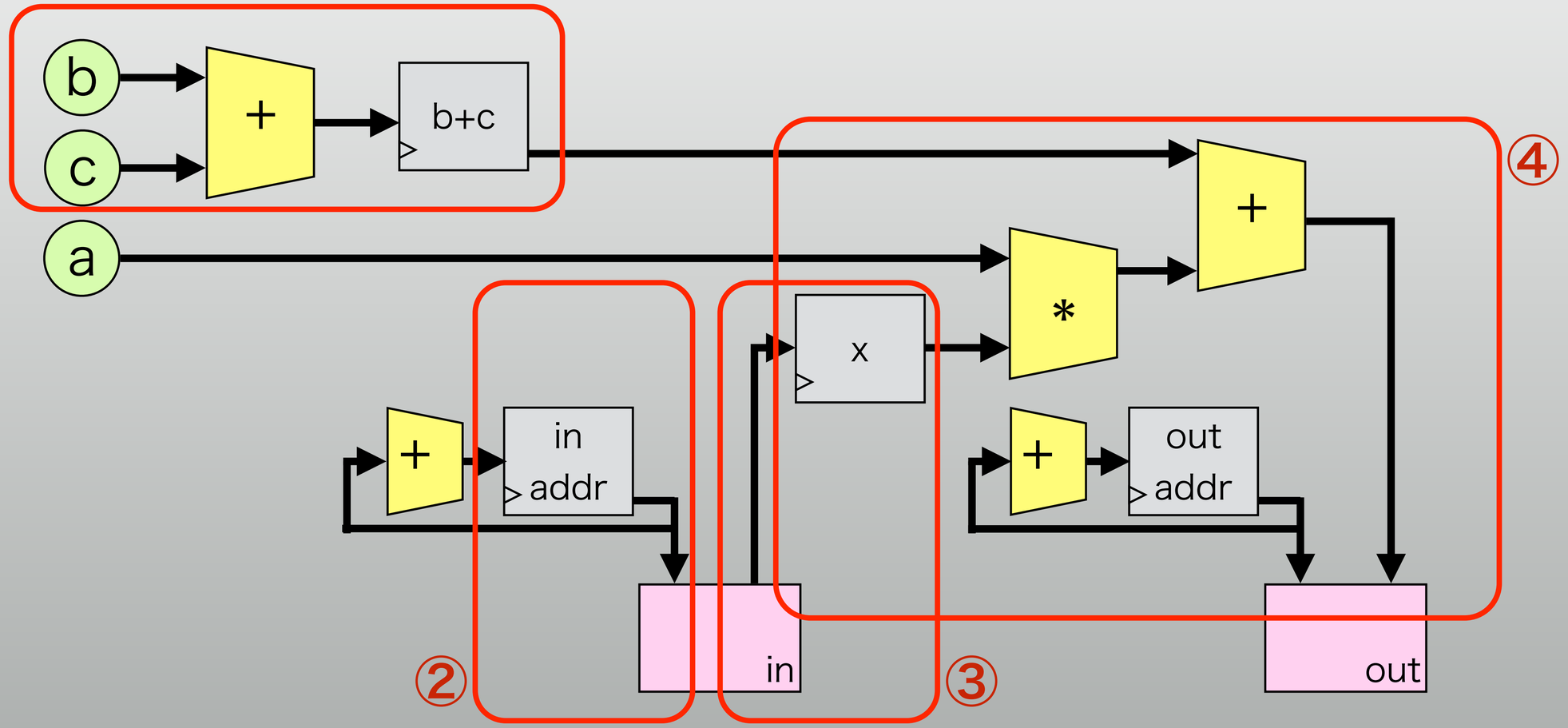
- Separate constant value calculations
  - Do only once before the loop begins

```
void foo( int in[3],
          char a, char b, char c,
          int out[3]) {

  int x, y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

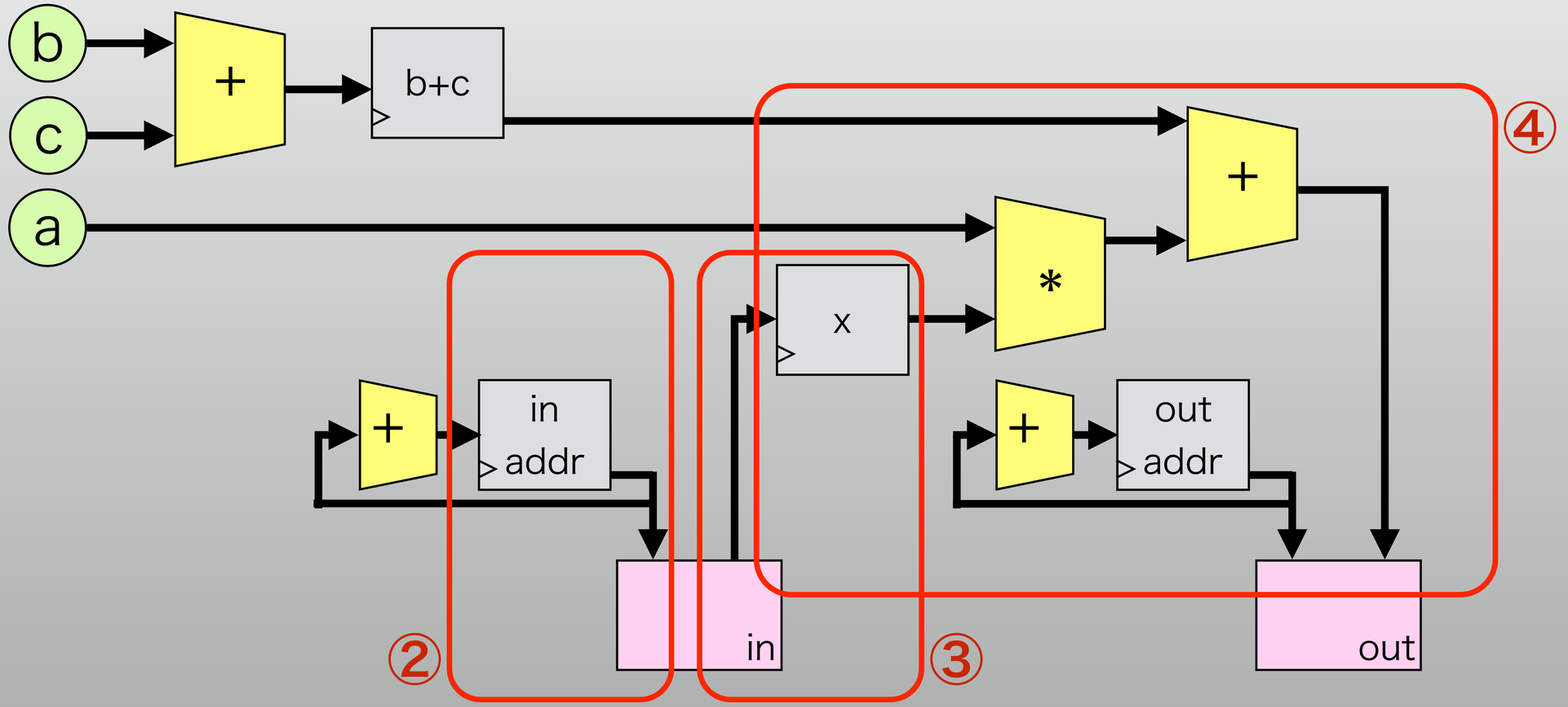
Xilinx UG902 (v2014.1)

Iteration 1

①: Preprocess

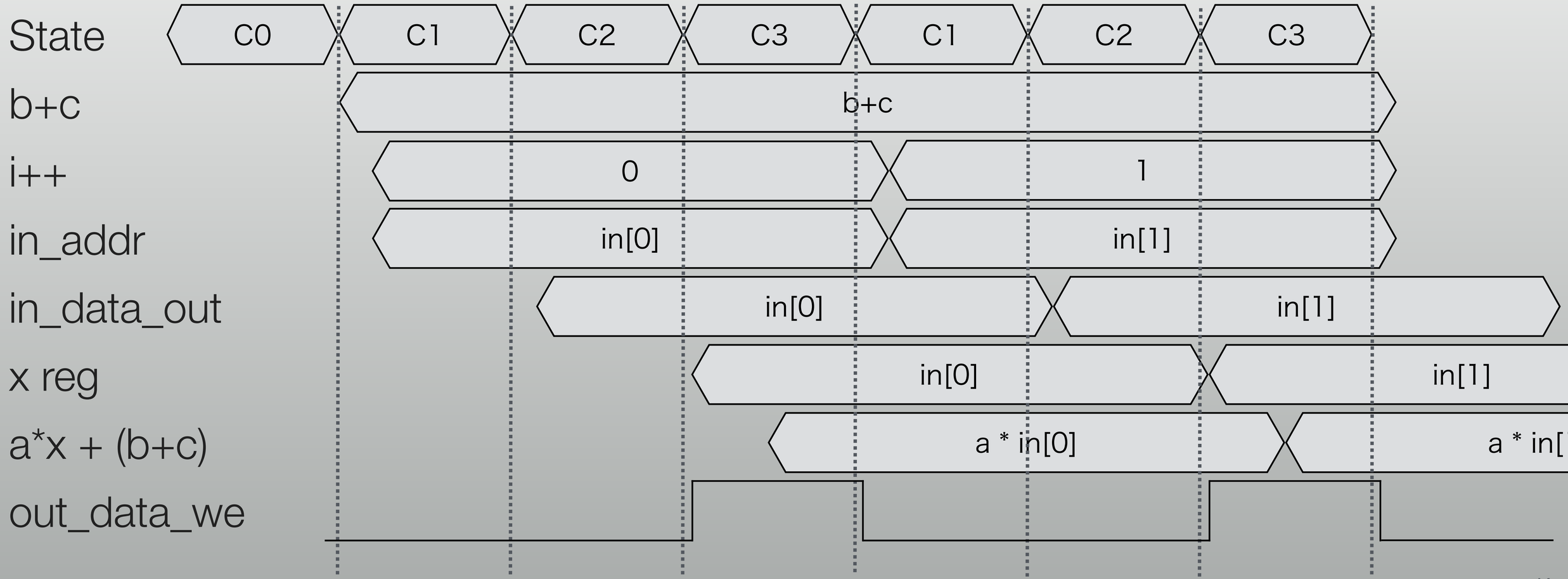


Iteration 2 and later



Control State

Control FSM is automatically generated



Control flow and FSM

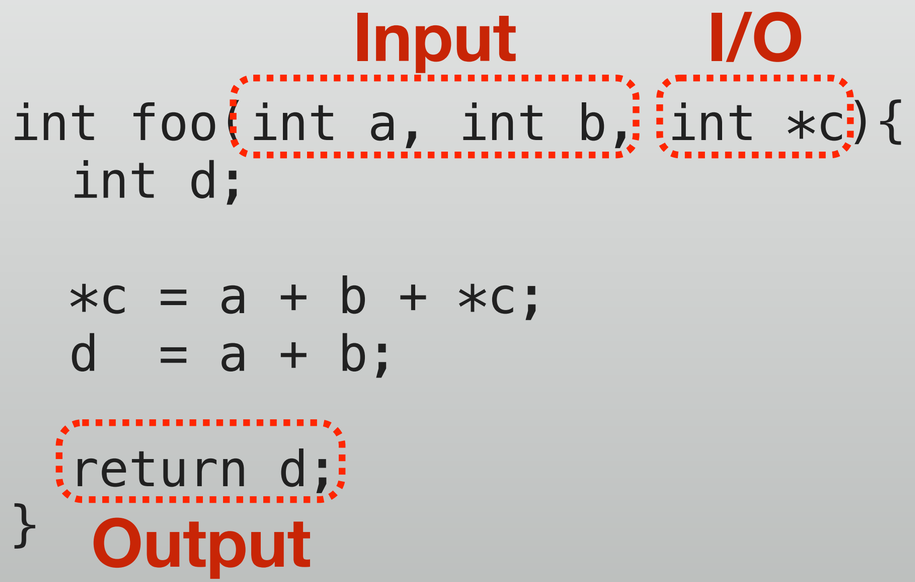
- Control flow = FSM states
  - I/O and branches changes state of the FSM
    - In other words, data path is switched by the FSM
- FSM state may not change every clock cycle: FSM may stay in a specific state (i.e., “Idle” state) for a long time

Interface synthesis

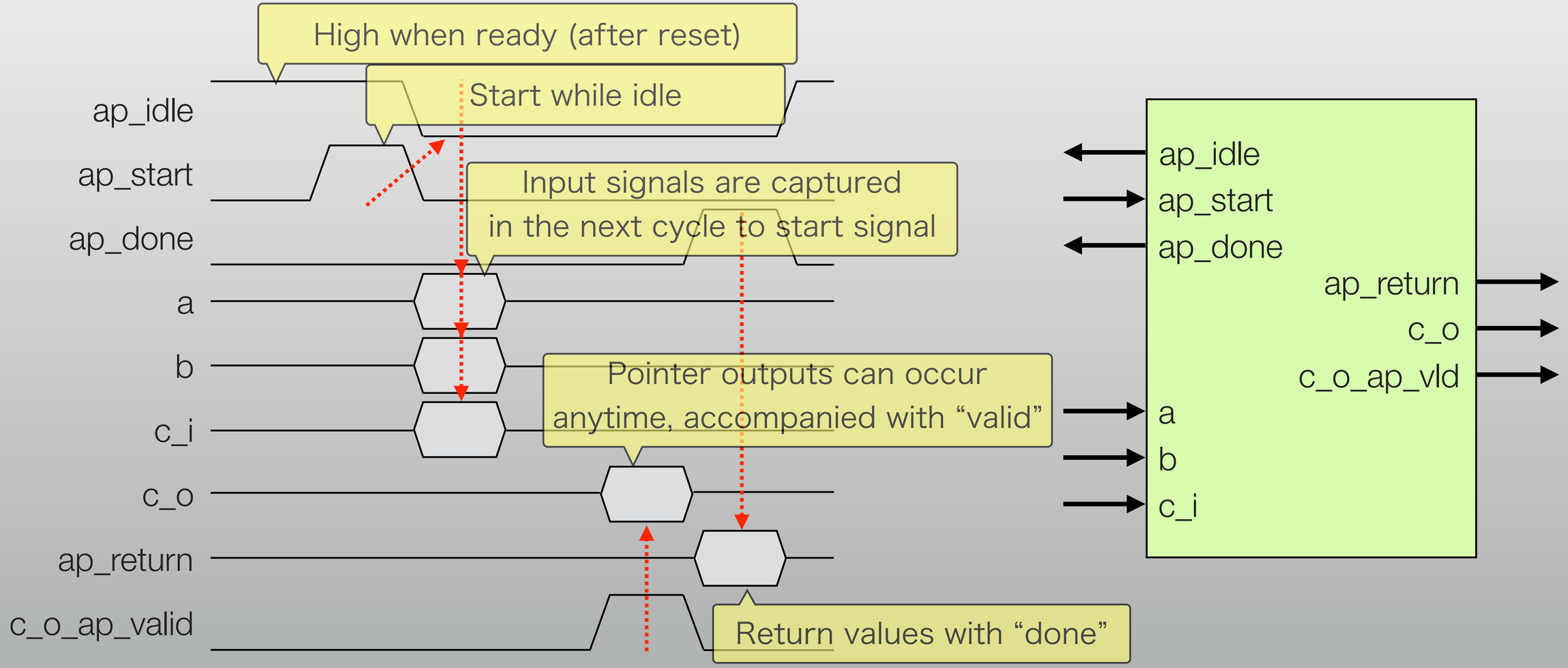
- C functions → How about the module ports in synthesized RTL?
  - In C, functions have 3 ways to perform I/O
    - Function arguments
    - Return valves
    - OS I/Os: printf(), gettimeofday() and others (not synthesizable)

Arguments, Return values and pointers

- Arguments are basically input
  - Pointer arguments can be output
- Return values are always output

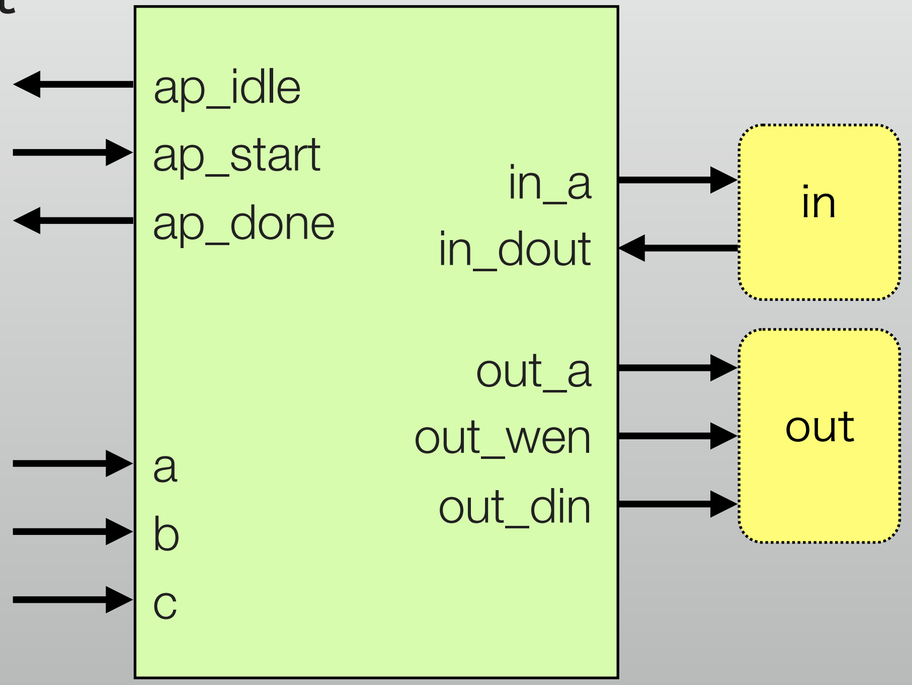


Basic module interface



BRAM I/F example

- BRAM I/F synthesized for array argument
  - BRAM is not included internally
  - Write before ap_start,
    Read after ap_done
- BRAMs are placed externally, this gives a great flexibility



More on Interface Synthesis

- Variety of interfaces are supported to meet various demands
  - Synthesized along coding style and directives
    - Directives given by #pragma or separate directive files
    - Refer tool's documentation: User Guide 902 for Vivado HLS

Optimization

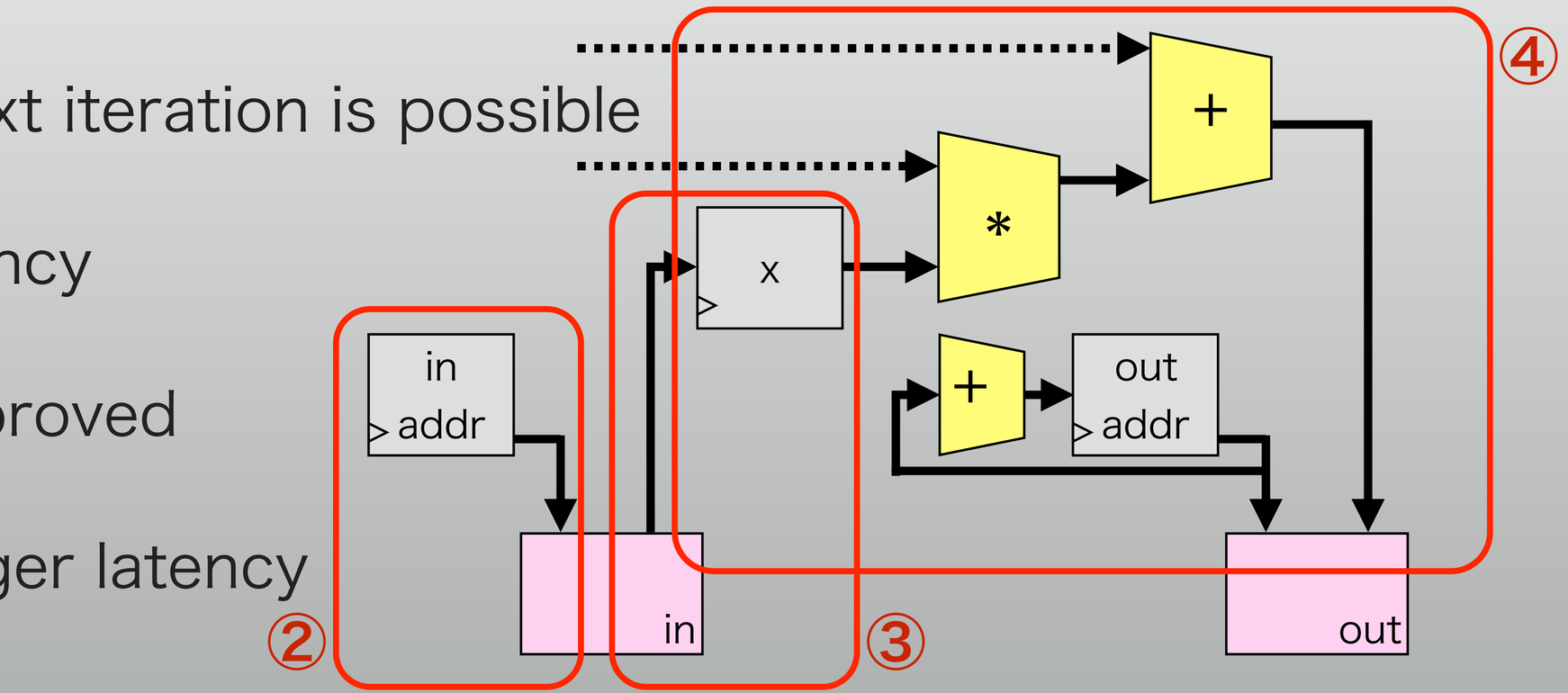
- In RTL design, resulting hardware has less variations
  - Because RTL description gives a rigid, clock-accurate model
- RTL generation in HDL has much more design options
  - Many possible designs from the same source code
    - This is good for fine-tuning of the design, but there's no "one-click solution"

Performance metrics

- 2 major concerns: area and speed
  - Area: How many LUTs, FFs and other blocks are used
  - Latency: clock cycles required between input and output
  - Interval: clock cycles required between input and next input

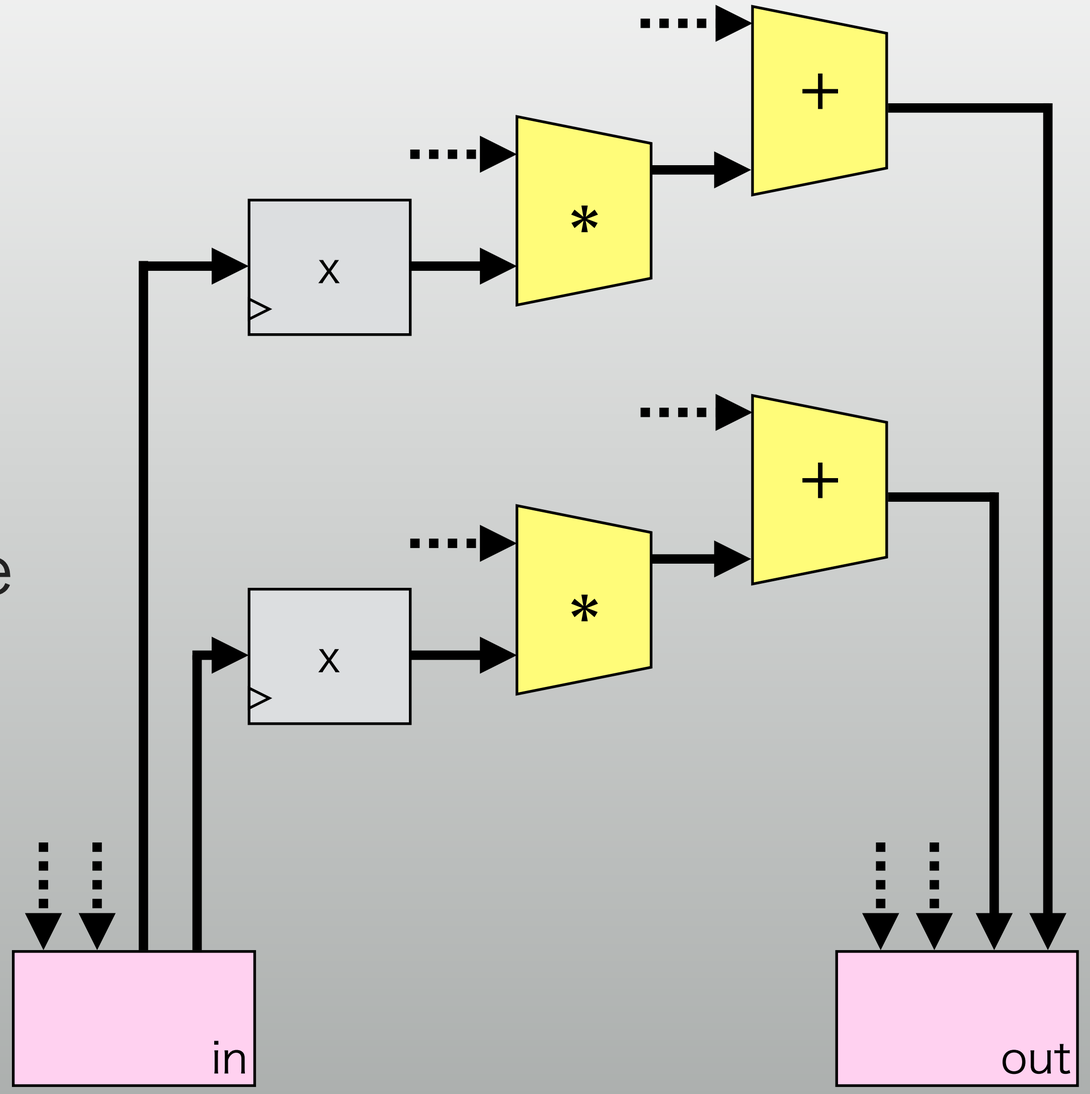
Loop Pipelining

- ②③④ in the previous example
  - On ③, ② for next iteration is possible
- No change in latency
  - Throughput improved
- Effective with longer latency



Duplication

- BRAMs are dual-ported
  - 2x pipelines gives 2x performance
  - Of course, 2x area is required



Source of difficulties

- Dependencies between loop iterations
  - This is really problematic: but appears in today's hands-on…
- Non-constant number of loop iterations
- Without these problems, loop are pipelineable easily

Optimization directive: pragma

- pragma in source files
  - Safely ignored by (software) compilers
  - Good for making many small tests

```
void foo ( int in[3],
           char a, char b, char c,
           int out[3]){
#pragma HLS RESOURCE variable=in core=RAM_1P

  int x, y;
  for (int i=0; i<3; i++){
#pragma HLS PIPELINE
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

Optimization directive: directive file

- Using separate directive file
  - Only labels in source code
  - Multiple directive files for multiple optimization strategies is possible

```
void foo ( int in[3],
           char a, char b, char c,
           int out[3]){
  int x, y;

add_loop:
  for (int i=0; i<3; i++){
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

```
set_directive_resource -core RAM_1P "foo" in
set_directive_pipeline "foo/add_loop"
```

C programming for HLS (1)

- Vivado HLS supports “normal” ANSI C and C++
  - Under several restriction, most syntax can be synthesized
    - Exceptions: pointers, recursions, memory allocations, OS I/Os…
  - main() for testbench, some function will be the “top-level module”
    - Everything are allowed in testbench

C programming for HLS (2)

- Still understandable / executable as a software
  - But it's "hardware description"
  - Fine-tuned software will be very insufficient with HLS: different optimization is required for HLS design

Example: Least Common Multiple

- Very simple algorithm to calculate LCM
  - With C test bench

```
int lcm(int a, int b){
  int x = a*b;

  // make a > b
  if (a<b){
    int tmp=a;
    a = b;
    b = tmp;
  }

  int r = a%b;
  while (r!=0){
    a=b;
    b=r;
    r=a%b;
  }
  return x/b;
}
```

C Testbench

- Just call the lcm() function
  - Show the result by printf()

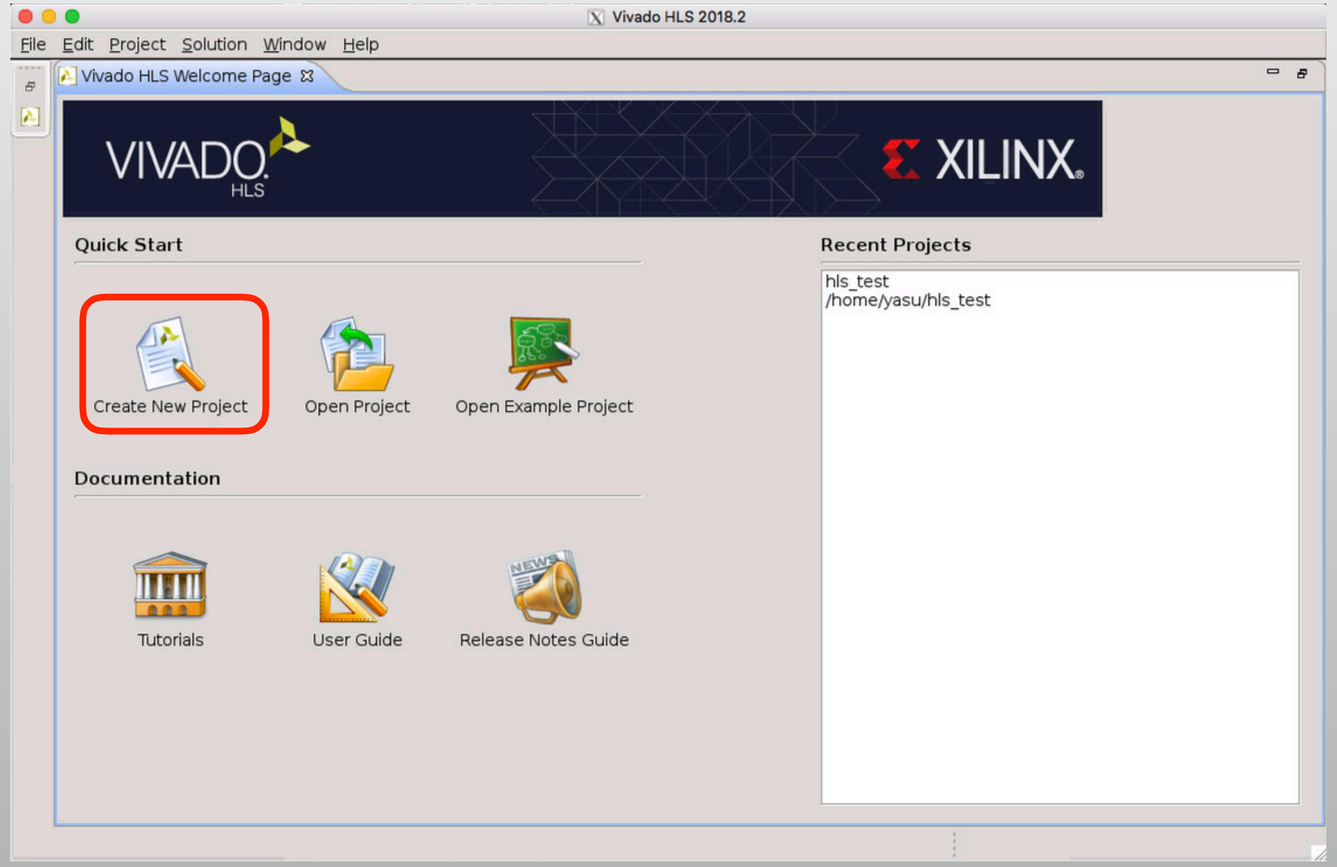
```
#include <stdio.h>

int lcm(int, int);

int main(){
  printf("2,3 %d\n",  lcm(2,3));
  printf("5,12 %d\n", lcm(5,12));

}
```

Vivado HLS design flow

- Write source code + testbench
  - Just compile and run to test as a software
- C Synthesis with Vivado HLS
  - RTL Co-Simulation to verify with C TB + Synthesized RTL
  - Try optimize along C Synthesis reports
- RTL Export → import to RTL design as an IP core

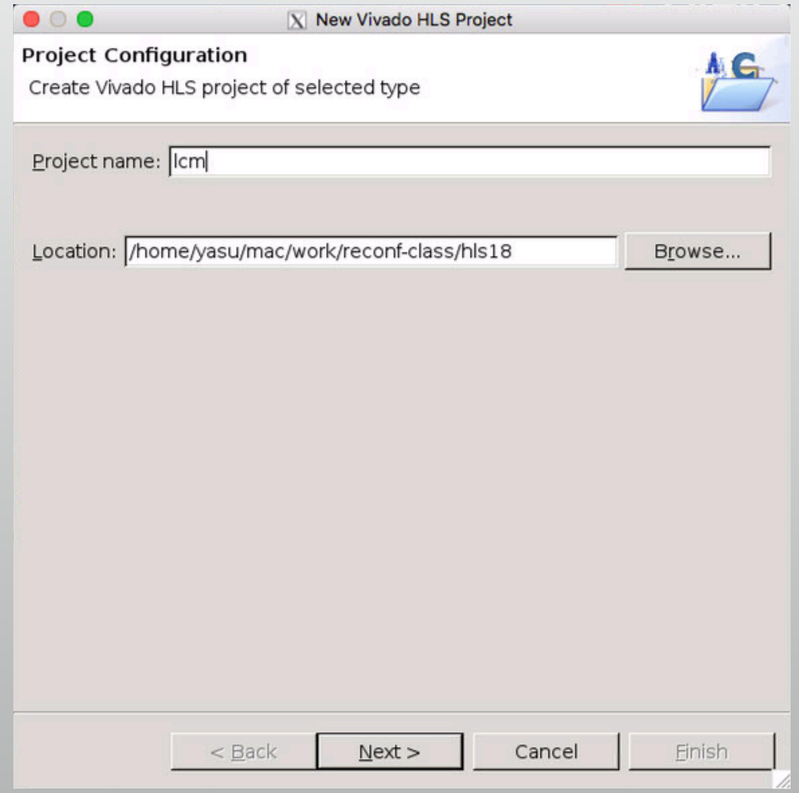
Create Vivado HLS project

- Launch Vivado HLS
  - Not “Vivado” for HDL design
  - Vivado HLS and Vivado is different, of course separate project folders are required



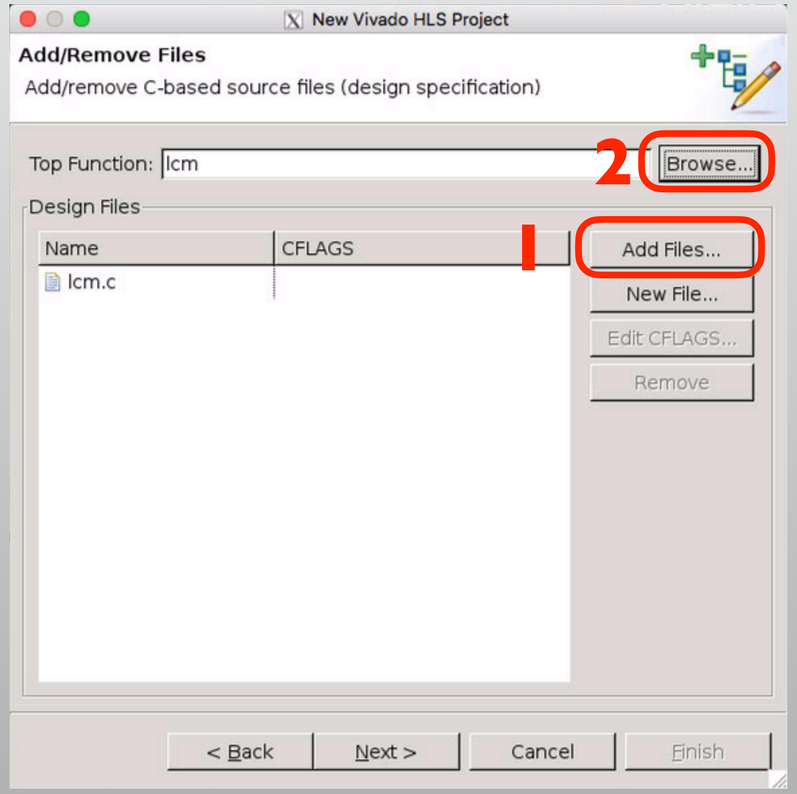
Specify project name+location

* “Location/project name” folder will be created



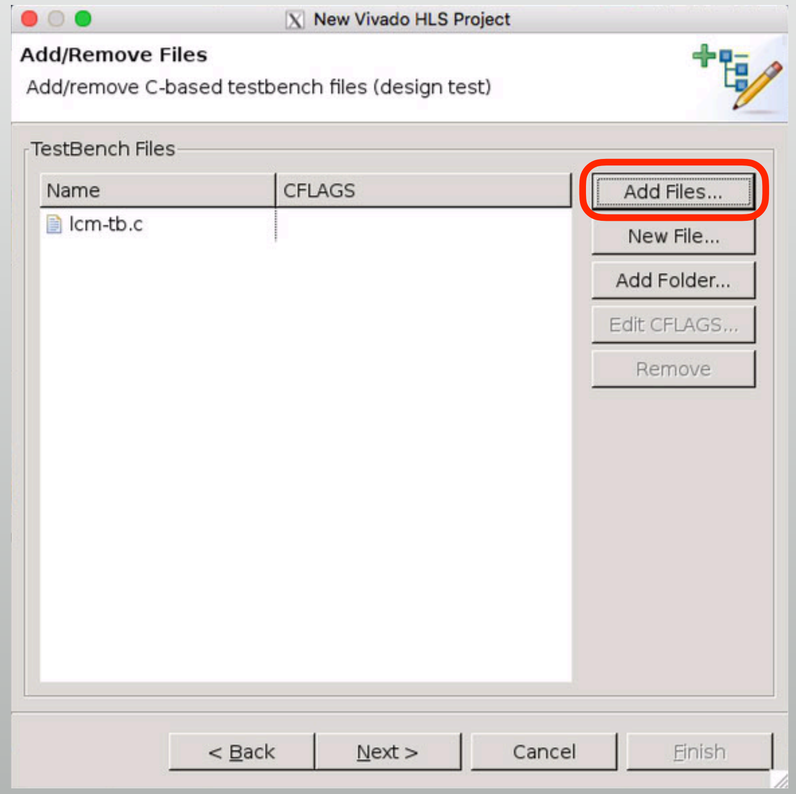
Add design file

- “Add Files” then choose the design file (lcm.c)
- Then set top function by using “Browse” button



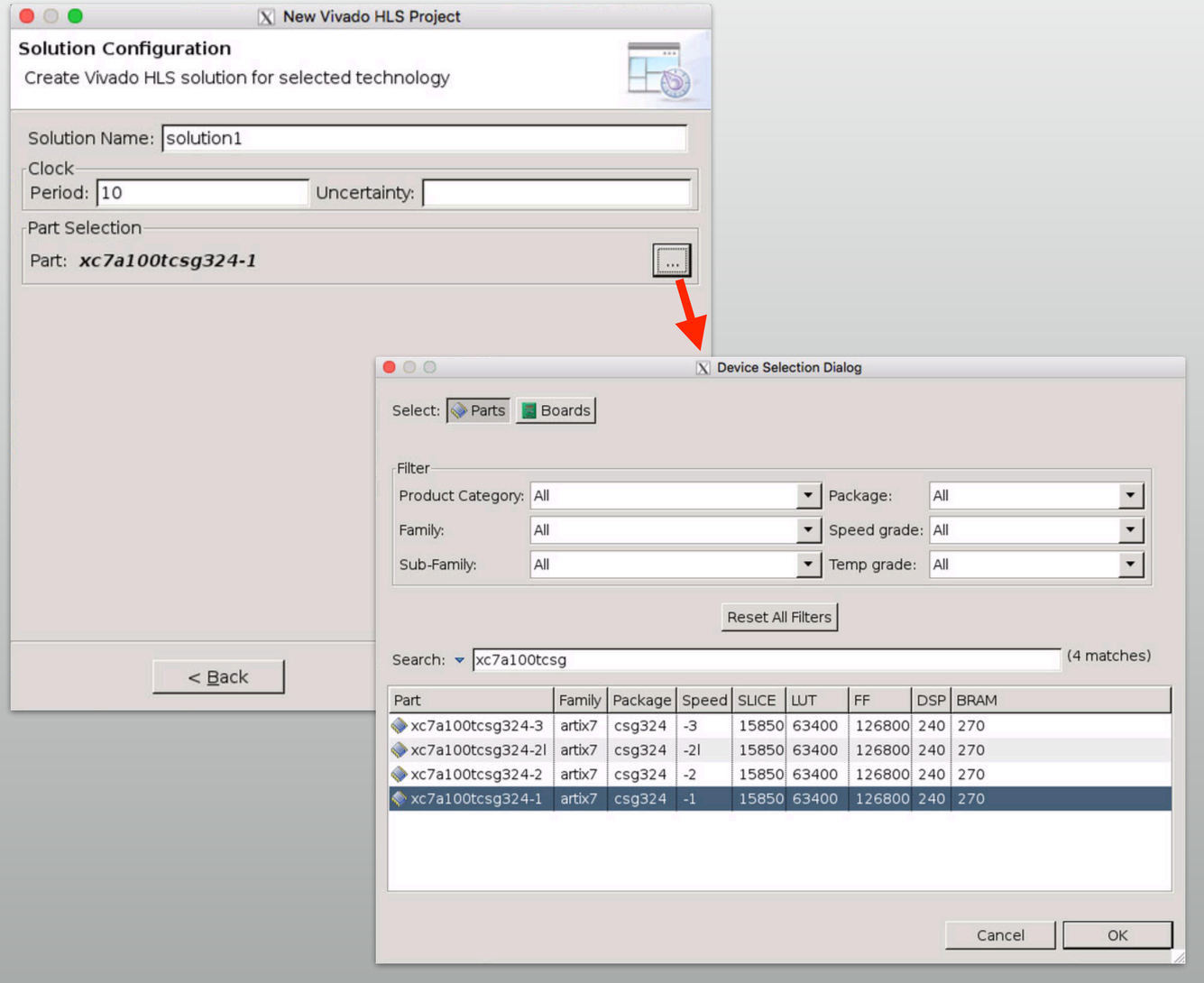
Add testbench

* Just “Add Files” then choose lcm-tb.c



Set Solution & Select device

- Default clock period is 10ns
  - No problem for Nexys4
  - Details for solution later
- Device is the usual one:
  - xc7a100tcsg324-1

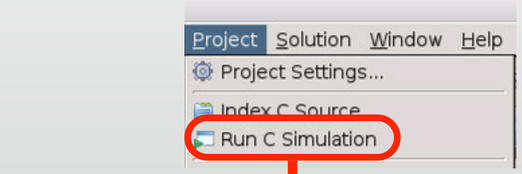


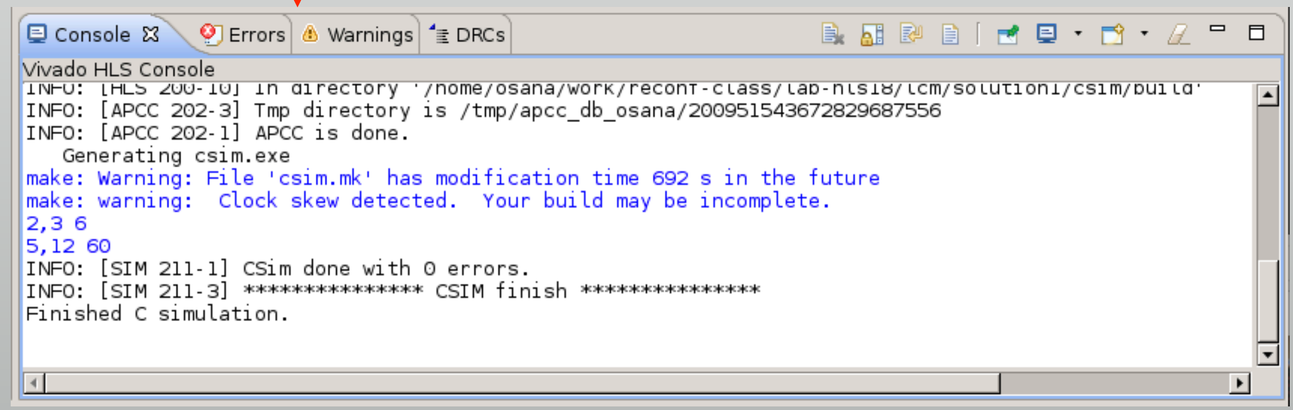
What’s the “Solution?”

- A optimization strategy set including:
  - Target clock frequency
  - Target device
  - Optimization directives in directive file (not by #pragma)
- Multiple solutions in single project is possible!

C simulation

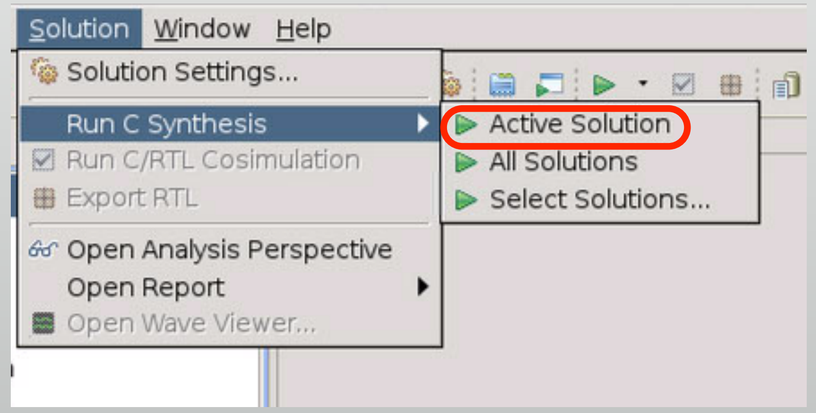
- Simulate as a software
  - Project → Run C simulation
  - Result will appear on the console





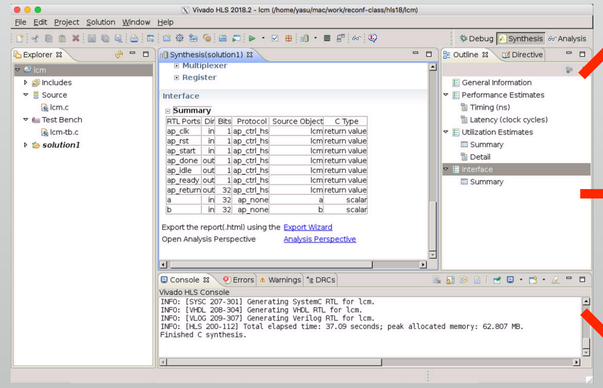
C synthesis

- Synthesis is done per solution basis
  - RTLs are generated



Synthesis report

- Things to be certain:
  - Latency/Interval
  - Interface
- Circuit size (Utilization) is undetermined until mapped in Vivado



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

	Latency			Initiation Interval		
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count
- Loop 1	?	?	36	-	-	?

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	121
FIFO	-	-	-	-
Instance	-	-	1182	714
Memory	-	-	-	-
Multiplexer	-	-	-	491
Register	-	-	235	-
Total	0	3	1417	1326
Available	270	240	126800	63400
Utilization (%)	0	1	1	2

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	lcm	return value
ap_rst	in	1	ap_ctrl_hs	lcm	return value
ap_start	in	1	ap_ctrl_hs	lcm	return value
ap_done	out	1	ap_ctrl_hs	lcm	return value
ap_idle	out	1	ap_ctrl_hs	lcm	return value
ap_ready	out	1	ap_ctrl_hs	lcm	return value
ap_return	out	32	ap_ctrl_hs	lcm	return value
a	in	32	ap_none	a	scalar
b	in	32	ap_none	b	scalar

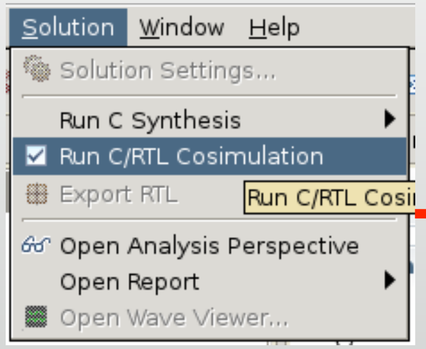
Reading the report

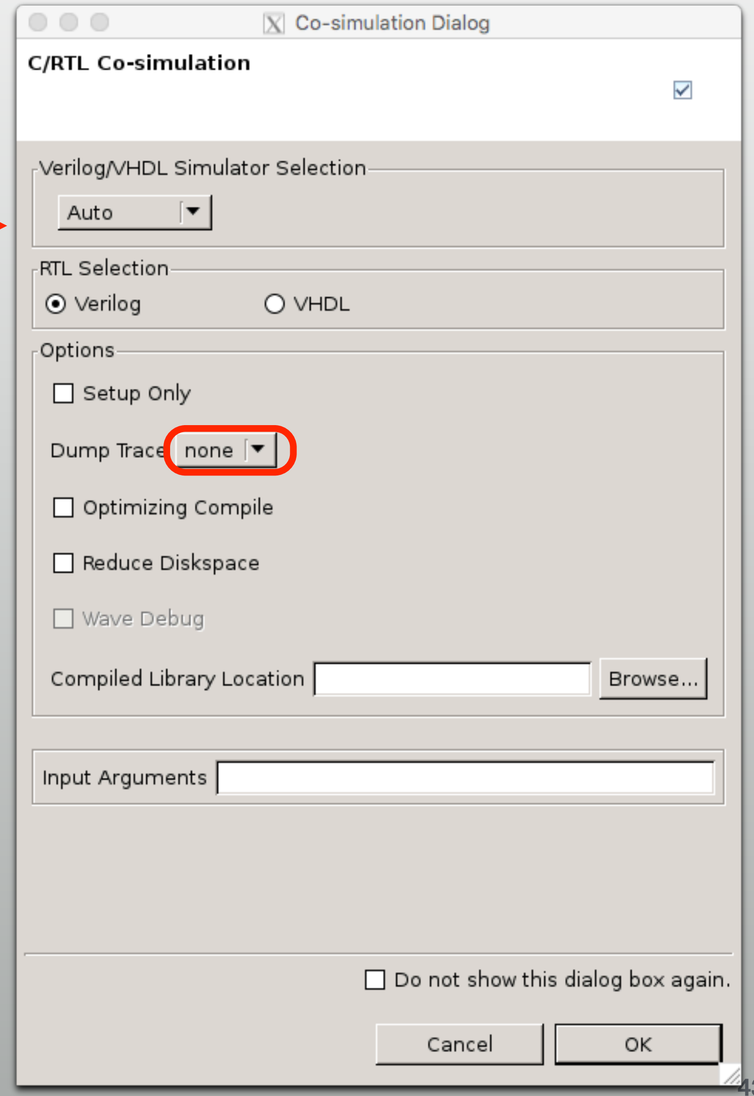
- Latency and Interval is “?”
  - Because there’s a “while” loop, the # iterations undetermined
  - Insert “#pragma HLS LOOP_TRIPCOUNT avg=10” in the while loop
    - then the latency is estimated in the report

```
11   int r = a%b;
12   while (r!=0){
13 #pragma HLS LOOP_TRIPCOUNT avg=10
14     a=b;
```

Launch Co-simulation

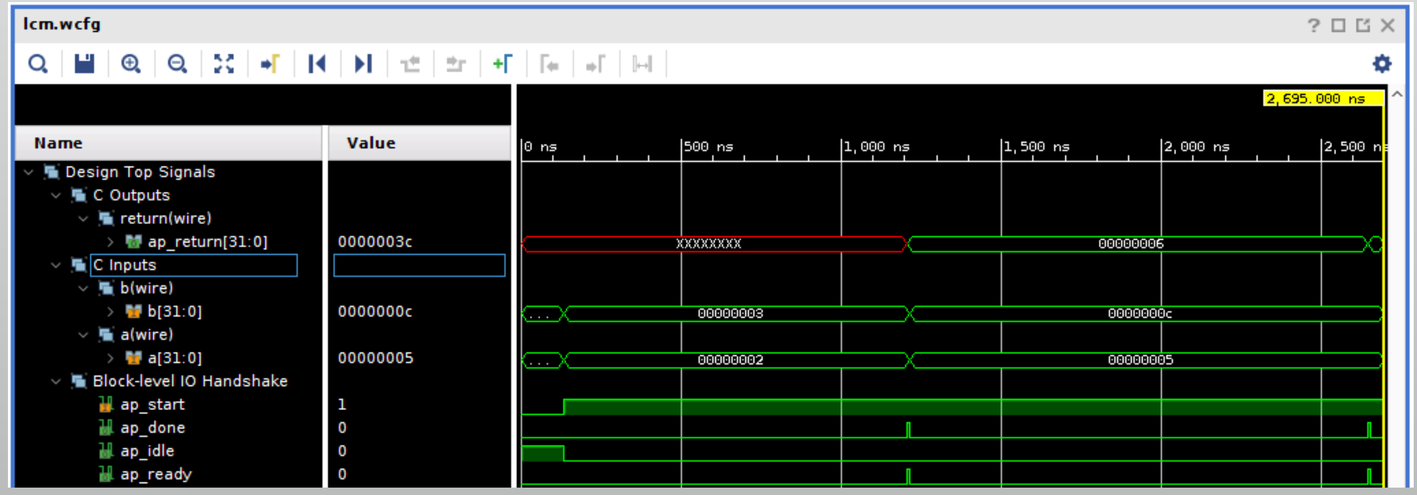
- C TB + Synthesized RTL
  - Set “Dump trace” to “All”
  - C TB is translated to HDL, results are compared





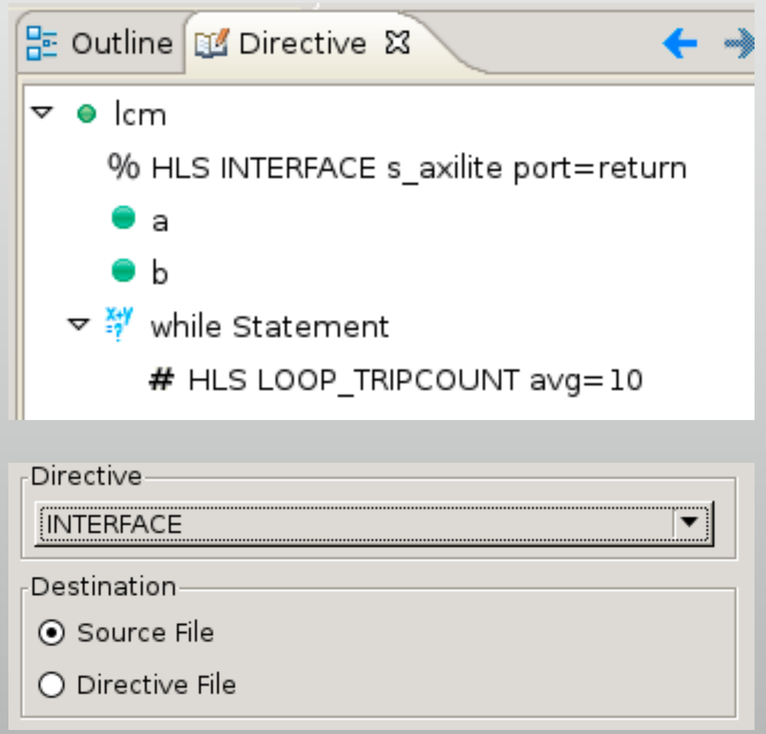
Co-simulation result

- printf() messages in Console
- Launch Waveform and see the signals
  - C Inputs/Outputs
  - Block Level IO Handshake: ap_{start,done,…}



Adding Directives

- Directives on functions, variables and loops
- In source file (#pragma) or in directive file (per solution)



Example: interface directive

- Set the lcm() function's "INTERFACE" to "s_axilite"
  - And also on argument a and b
  - Synthesized interface is gathered into single AXI slave interface

Analysis View

- Control state and dependency graph
  - Can refer source code