再構成型アーキテクチャ特論　(9)

高位合成 (1回目)

今回の内容

* 高位合成
  * C / C++ / Java や System C などから RTL を生成
  * OpenCL へ対応しているベンダもある

高位合成ツールの例

* 対象デバイスを選ばないもの
  * Impulse C (Impulse), Cyber WorkBench (NEC)
* 特定デバイスへ向けたもの
  * Vivado HLS (Xilinx)
* 特定プラットフォーム専用のもの
  * Carte (SRC: C/Fortran), MaxCompiler (Maxeler: Java)

高位合成のよいところ

- HDL では書くのが難しいアプリケーションを簡単に書ける
  - 基本的にソフトウェアとして記述・検証ができる
    - シミュレータを使わないので動作検証が高速
  - アルゴリズムの修正が容易
- 設計のトレードオフをどうとるかを簡単に試せる (性能 vs 面積 など)

高位合成が使われるよくある場面

- 金融: 市場の動向を早く読むのが重要
- 信号処理: 難しいディジタルフィルタを高位合成で実装
  - Matlab からのデザインエントリはわりと昔からできます
  - 画像処理とかにも有効

高位合成のつらいところ

* 速いとは限らない
  * 回路が (とても) 大きくなる場合も多い
* なんでも書けるわけではない
  * たとえば正確なタイミングの必要な外部インタフェイスなどは困難
  * HDL では難しいアルゴリズムの部分を高位合成するのが正解

高位合成ツールの基本的な仕事

* 制御フローの抽出
  * 条件分岐、ループ
* データフローの抽出
  * 基本的に Basic Block に対して行う
  * スケジューリングとバインディング

```
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;
  }
}
```

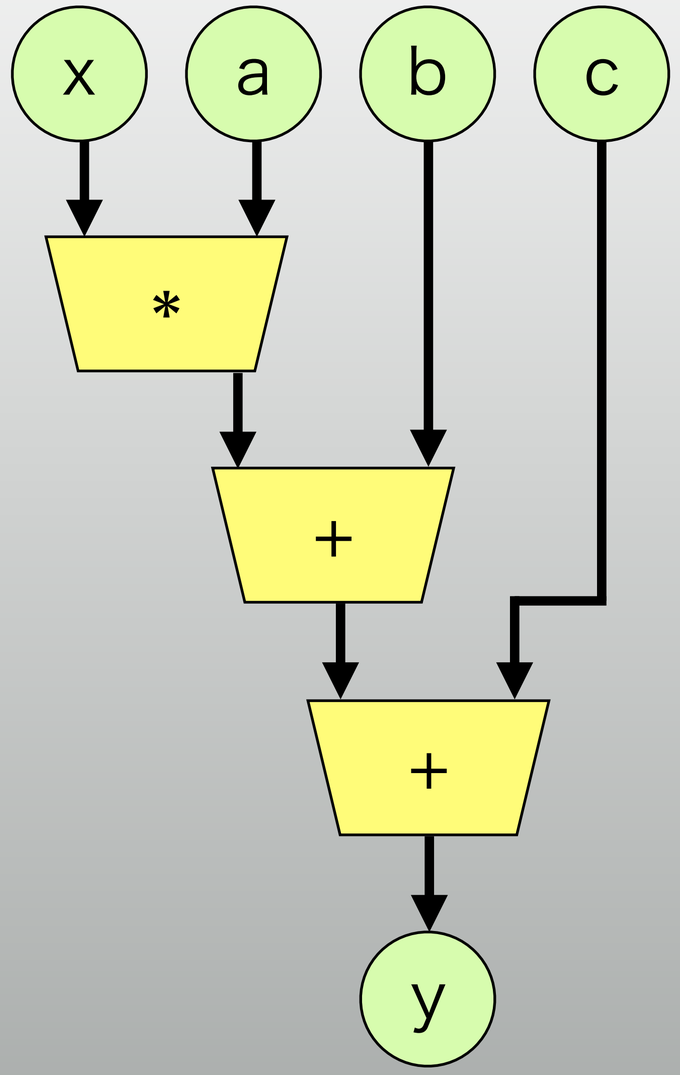
Basic Block

ループ

Xilinx UG902 (v2014.1)

データフローの抽出

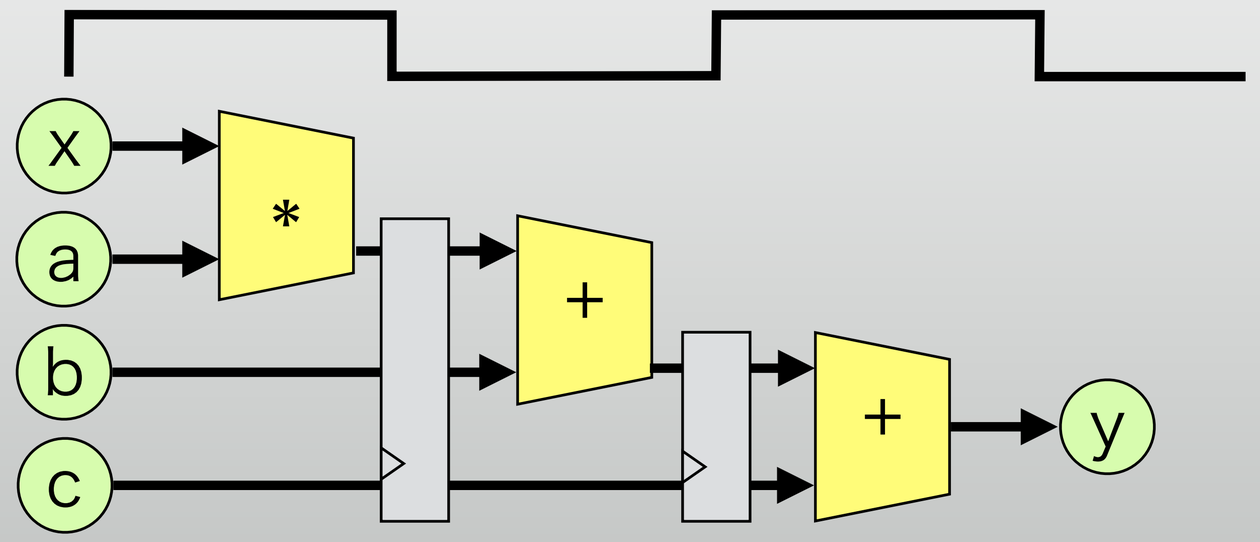
```
int foo(char x, char a,
        char b, char c){
  char y;
  y = x*a+b+c;
  return y;
}
```



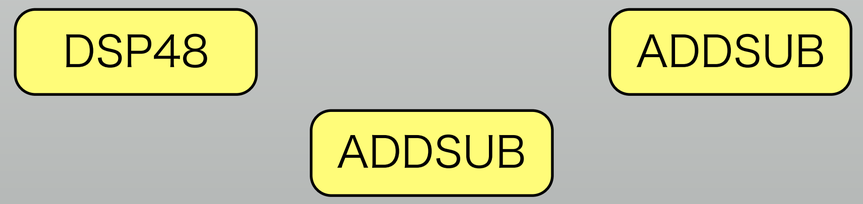
Xilinx UG902 (v2014.1)

スケジューリングとバインディング

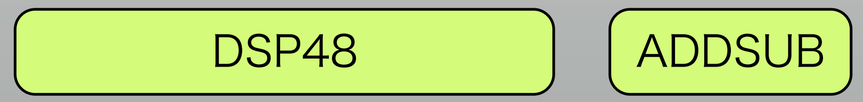
Scheduling



Initial Binding



Target Binding



制御フローの抽出、の前に入出力

* 引数がスカラ値の場合、普通の入出力ポートに
* 引数が配列の場合、BlockRAMインタフェイスに
  * 配列はメモリ経由で受け渡し
* 詳しくはあとで

```
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)

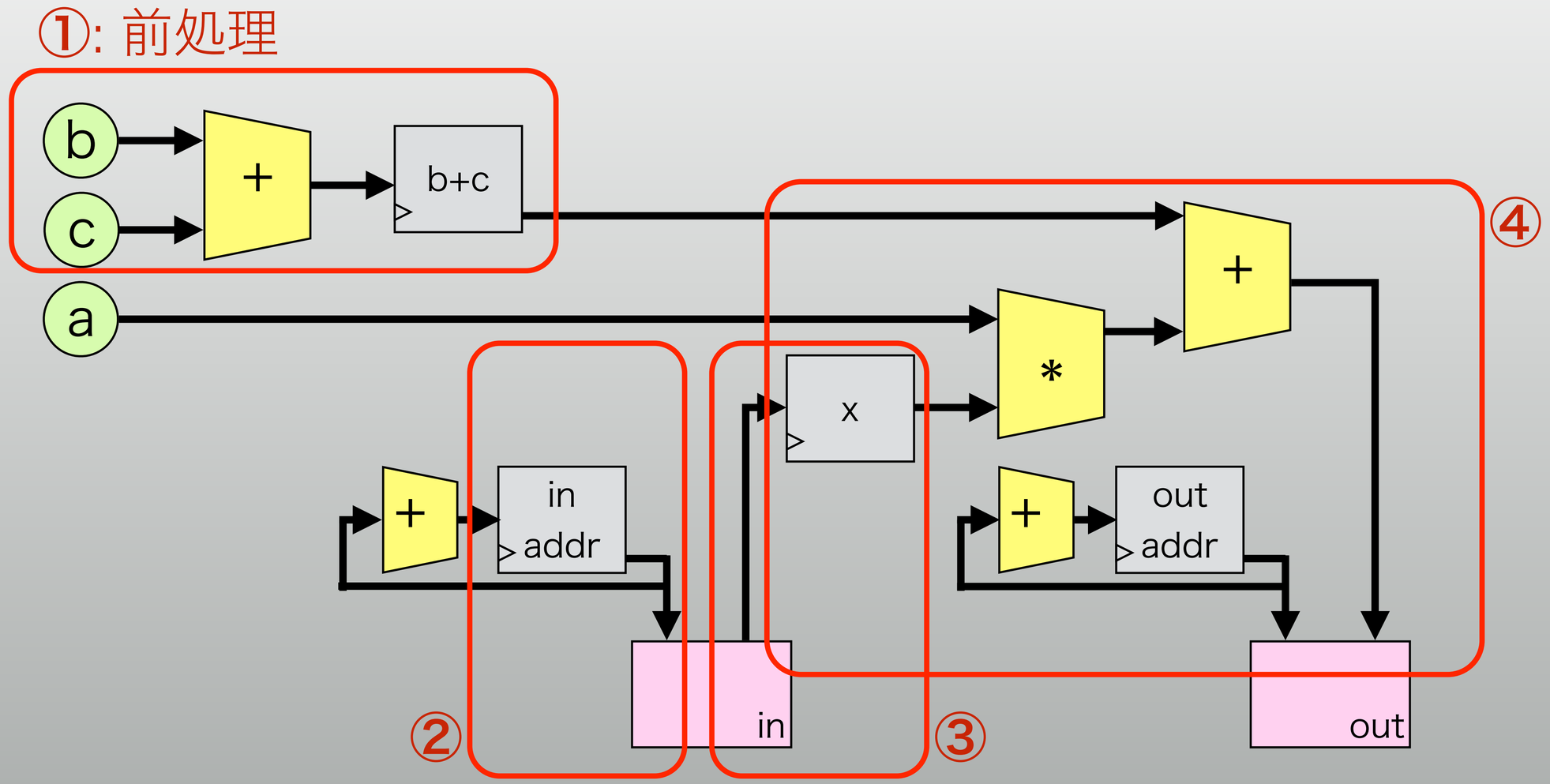
ループの場合は最適化も重要

* 一度だけやればよいことを分離
  * ループの処理をする前に行う

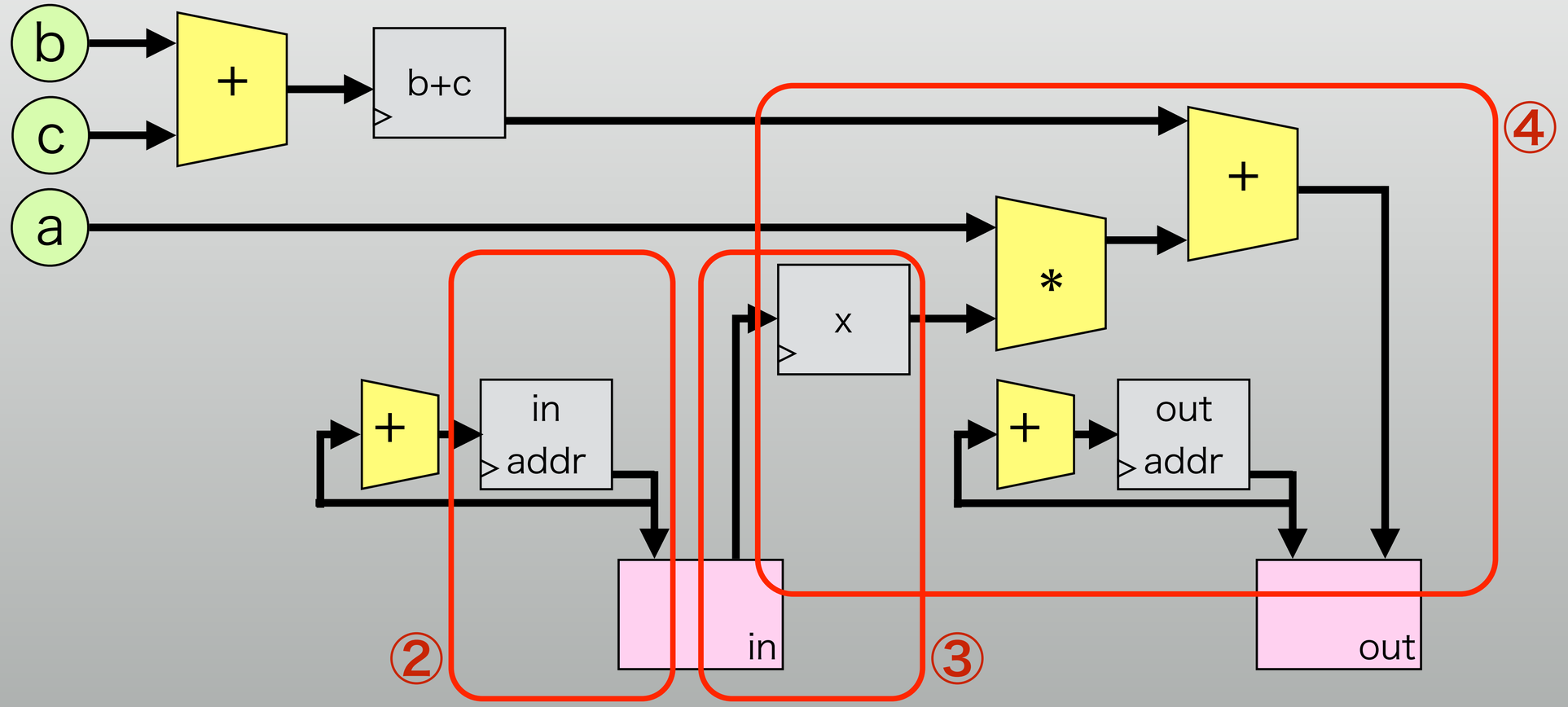
```
void foo( int in[3],
          char a, char b, char c,
          int out[3]) {

  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)

ループ1周目

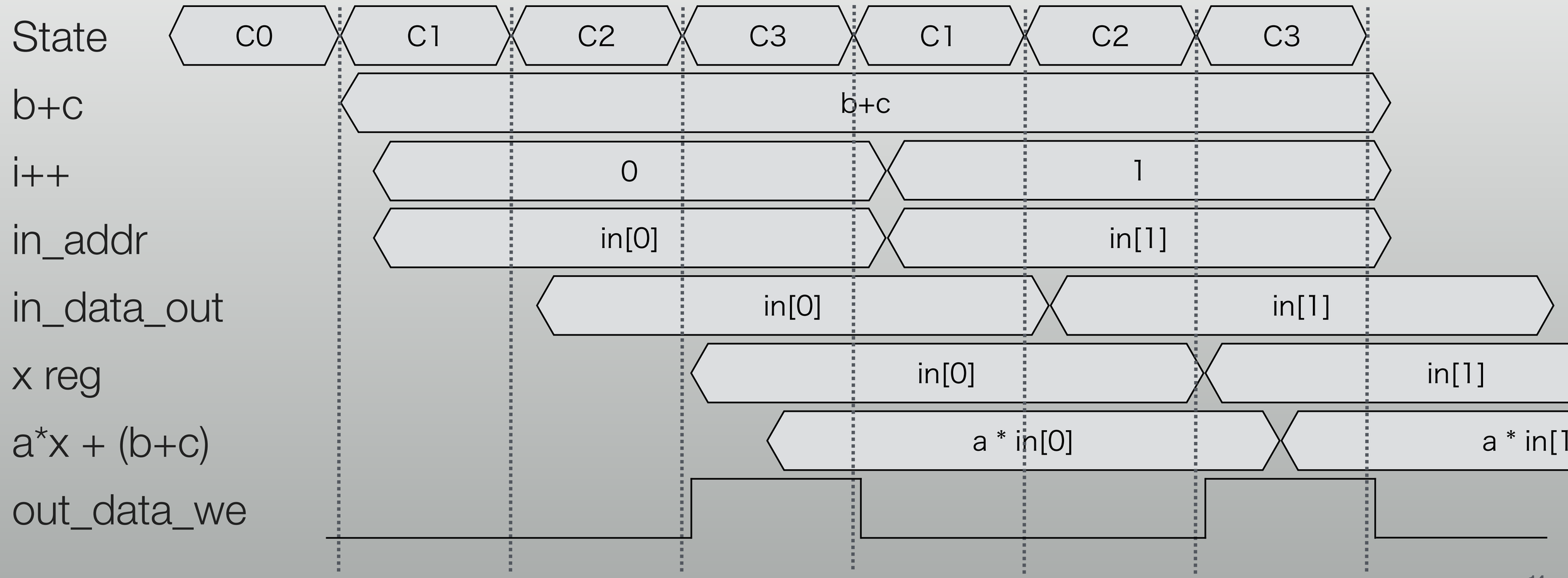


ループ2周目〜



Control State

I/O 信号制御とかに FSM (Control State) が使用される



制御フローとステートマシン

- 全体の流れはステートマシンが制御
  - 入出力や条件分岐でステートが変化する
    - データパスが変化するところ、と考えてもよい
- 各ステートの長さは必ずしも1クロックサイクルとは限らない

インタフェイス合成

* C で書いた関数 → RTL になったらモジュールのポートは？
  * C の関数の入出力は３つ
    * 関数の引数
    * 関数の返り値
    * OS による入出力: printf() や gettimeofday() など (合成できない)

引数、返り値、ポインタ

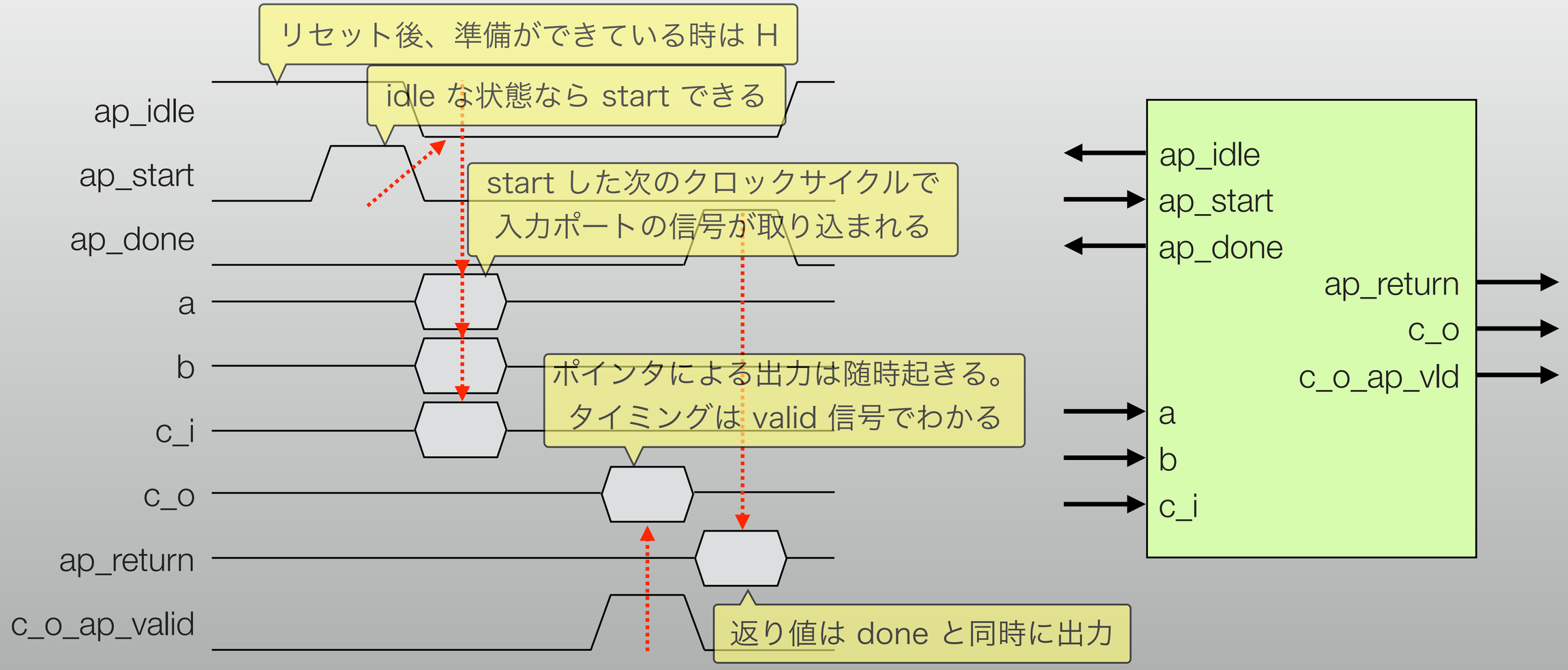
- 引数は基本的に入力
  - ポインタの場合は出力にもなりうる
- 返り値はいつも出力

入力　入出力

```
int foo(int a, int b, int *c){
  int d;

  *c = a + b + *c;
  d  = a + b;

  return d;
}
```

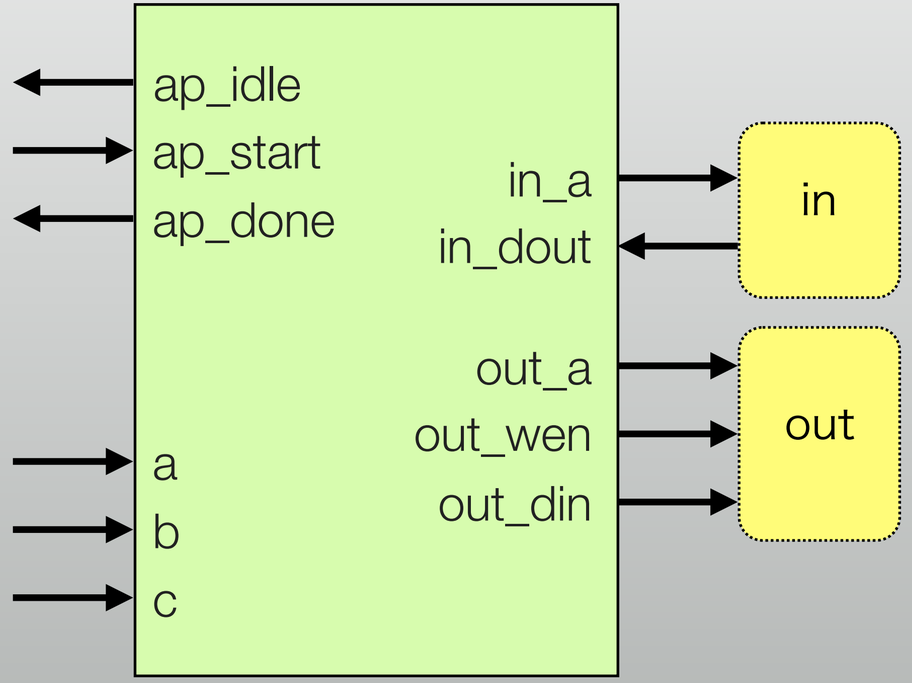
出力

基本的なモジュールインタフェイス



メモリを使う例 (さきほどのコードの場合)

* 引数に配列があるとメモリ I/F ができる
  * メモリは外部に配置
  * start 前に書き込み、done 後に読み出し
* メモリブロック自体は外部にあるので自由度が高い



インタフェイス合成

- 用途によっていくつものバリエーションがありうる
  - 必要に応じて選択できる
  - どういうのが使えるかは高位合成ツール次第
    - Pragma などを書いて指示する (後述)
    - 詳しくはマニュアルを

最適化

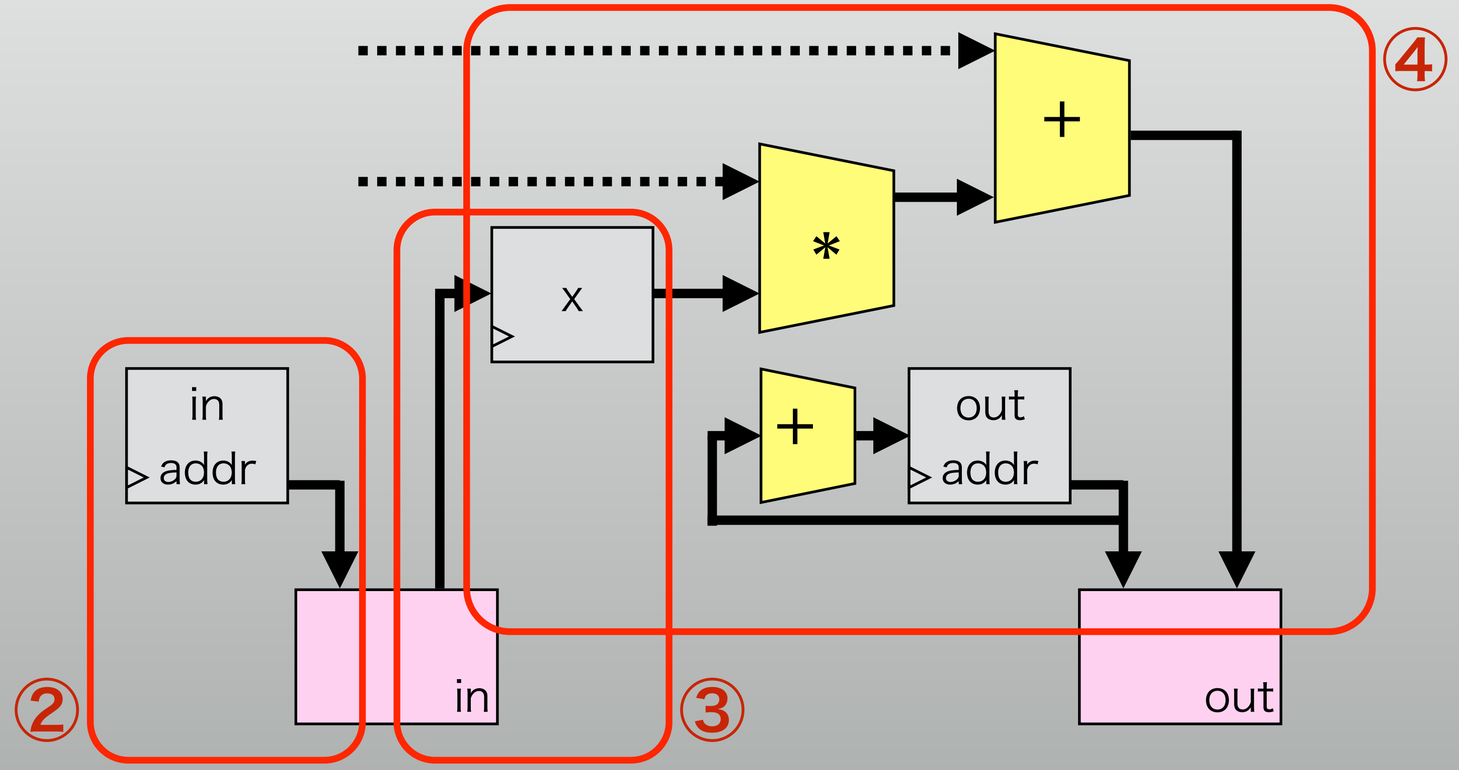
* RTL 設計では HDL 記述から生成される回路は大きくは変わらない
  * RTL 記述は少なくとも clock accurate であるため
* 高位合成で RTL を生成する際にはいろいろな選択肢があり得る
  * 同じコードでいろいろ試せるのが高位合成の大きなメリット
  * いろいろ試してみなければうまくいかない、ということでもある

性能指標

- 大きく3つ
  - Area: LUT とかをどれくらい使うか
  - Latency: データを入力→結果が出るまでのクロックサイクル数
  - Interval: データを入力→次のデータを入力できるまでの、それ
    - Latency より Interval は小さくも大きくもなりうる

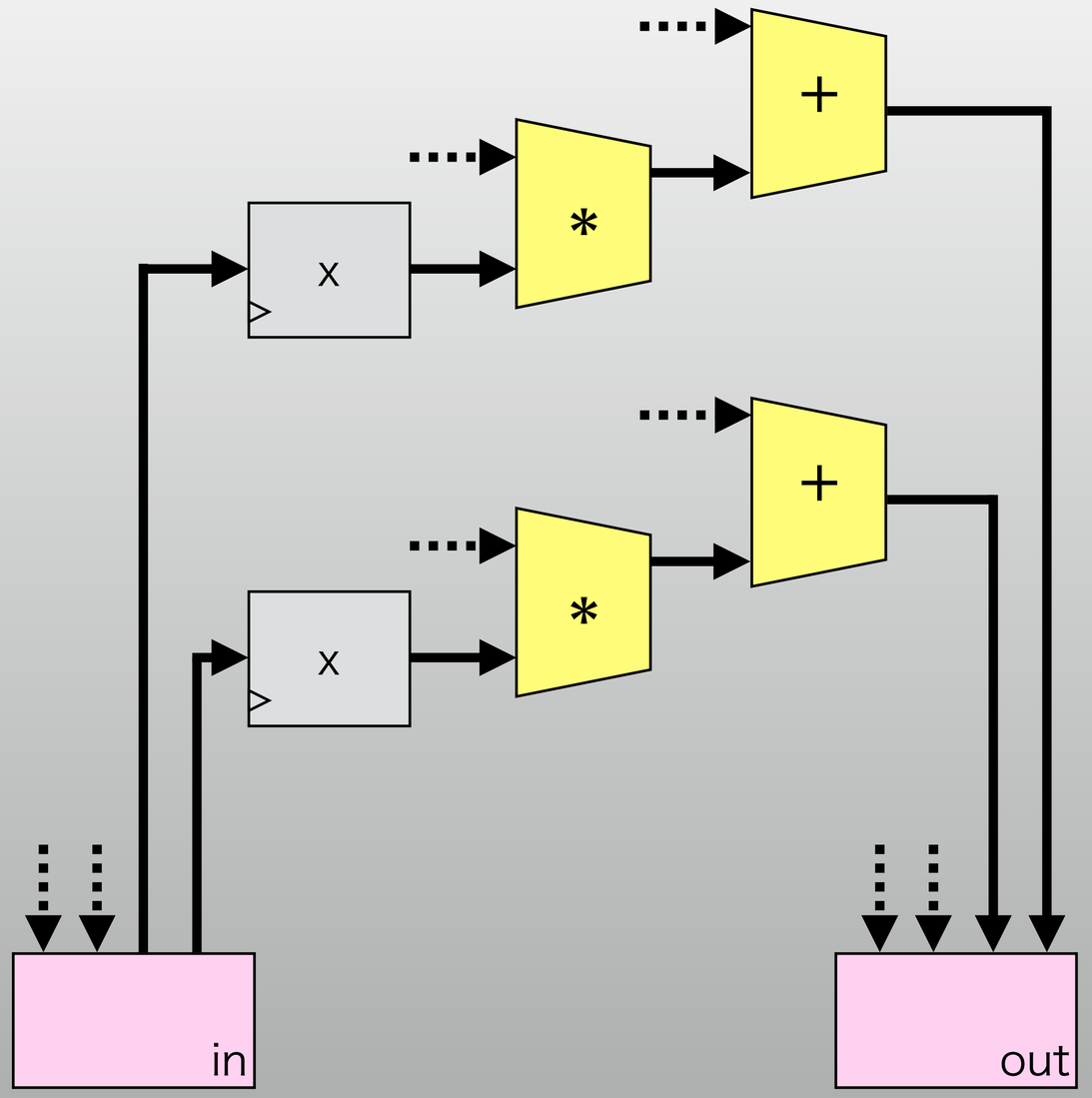
パイプライン化

- さきほどの②③④
  - ③がはじまったら次の②
- レイテンシは変わらない
  - スループットは向上
- レイテンシが長いほど効果大



並列化

- メモリはデュアルポート
  - 演算回路を2つ作れば2倍速!
  - パイプライン化と合わせると強力
- ただし面積が必要



うまく最適化できないもの

* ループの各 iteration に依存性があるもの
  * これから今回の例題がまさにそれ…
* ループの反復の回数が決まっていないもの
* そうでなければループはわりと最適化しやすい

最適化の指示: pragma

* ソースコードに pragma を記述
  * 合成時以外は無視されるので動作検証には影響しない

```
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;
  }
}
```

最適化の指示: directive file

- Directive file で別に指示する
  - ソースコード内には必要に応じてラベルを記述
  - Directive file にまとめて最適化指示を記述

```
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"
```

高位合成のための C プログラミング

- Vivado HLS は「ふつうの」ANSI C をサポート
  - gcc で動くコードは前述の制約を満たしていれば RTL に変換できる
  - テストベンチは主に main() に
    - 任意の関数をトップレベルモジュールとして RTL が生成できる
    - それ以外の部分には printf() でもなんでも書くことができる

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()

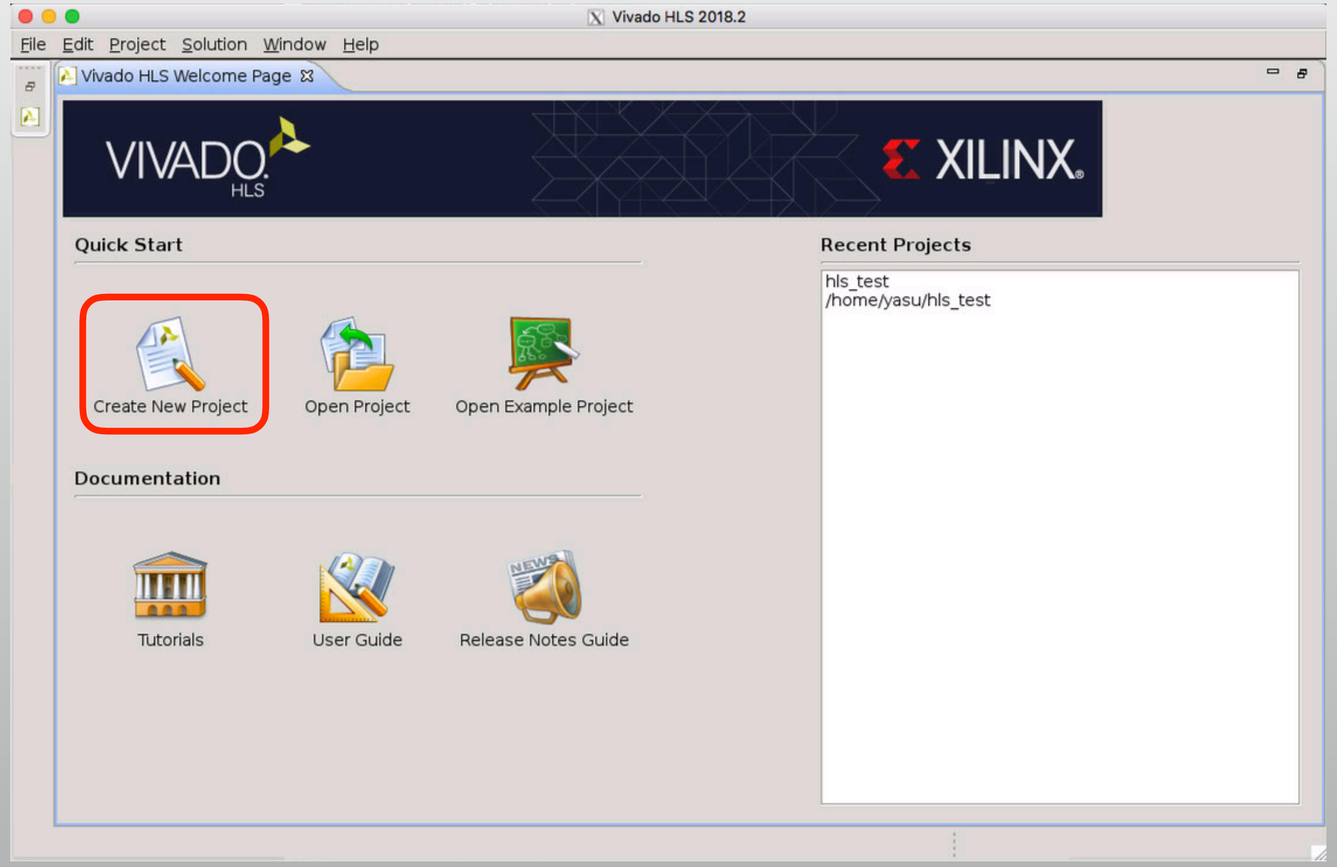
```
#include <stdio.h>

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

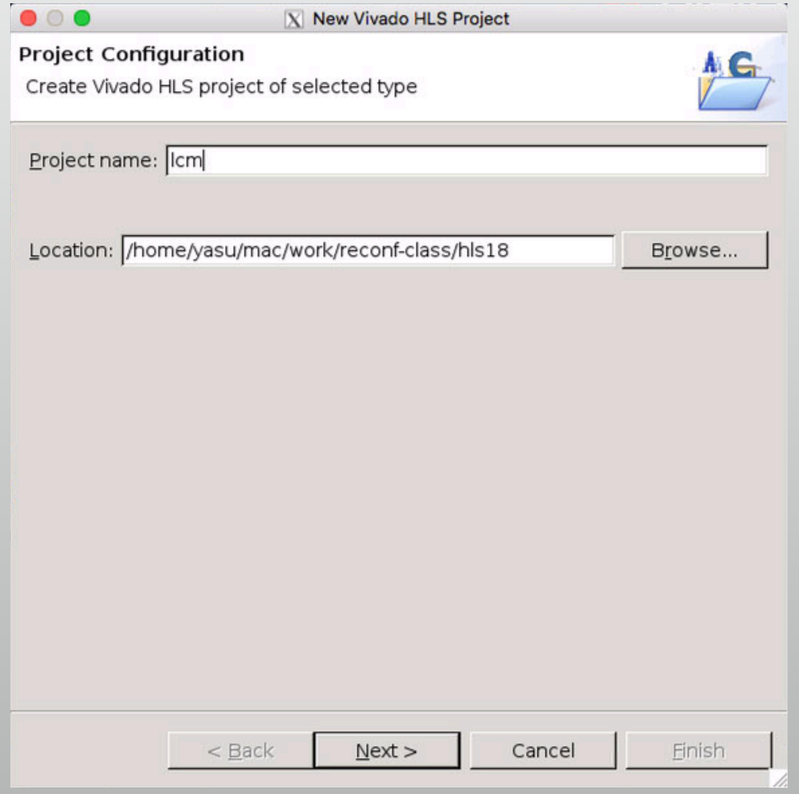
Create Vivado HLS project

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



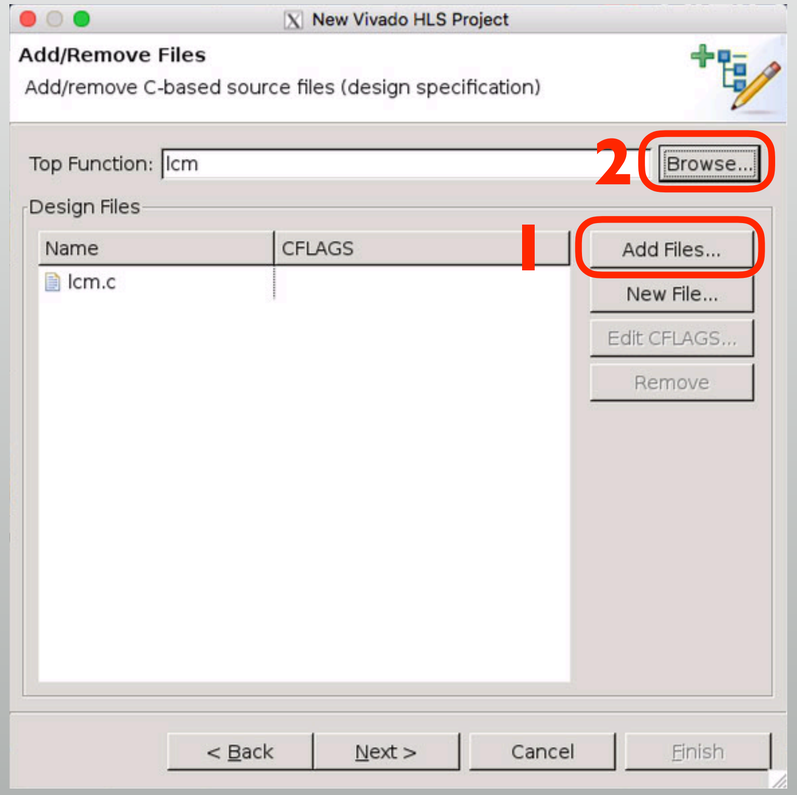
Specify project name+location

* “Location/project name” folder will be created



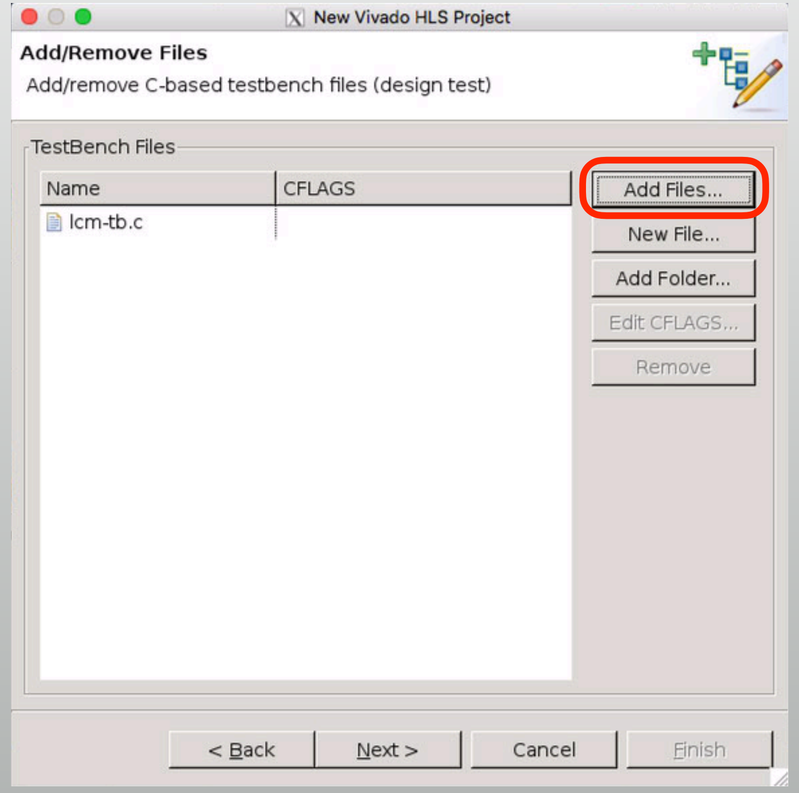
Add design file

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



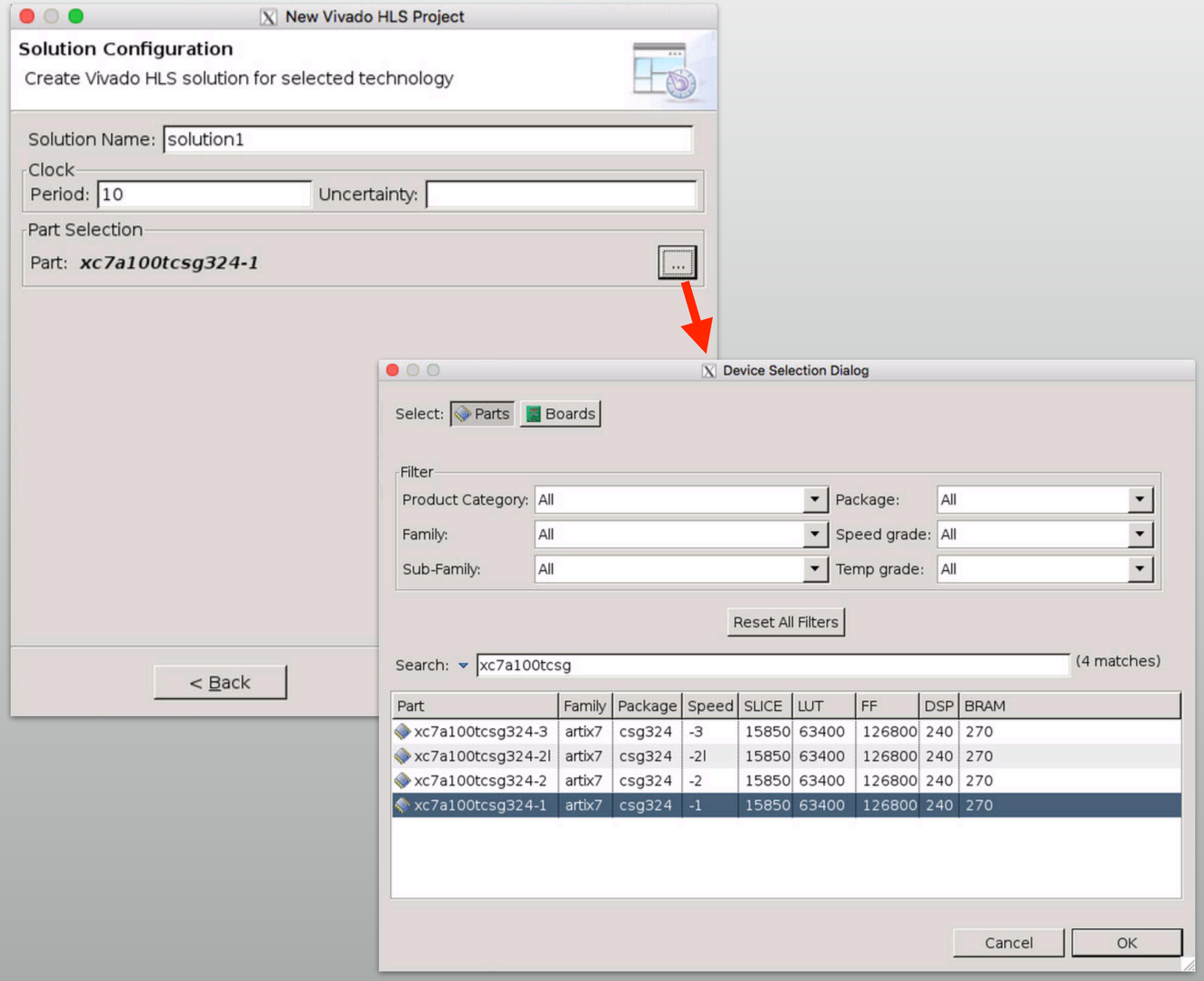
Add testbench

* 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

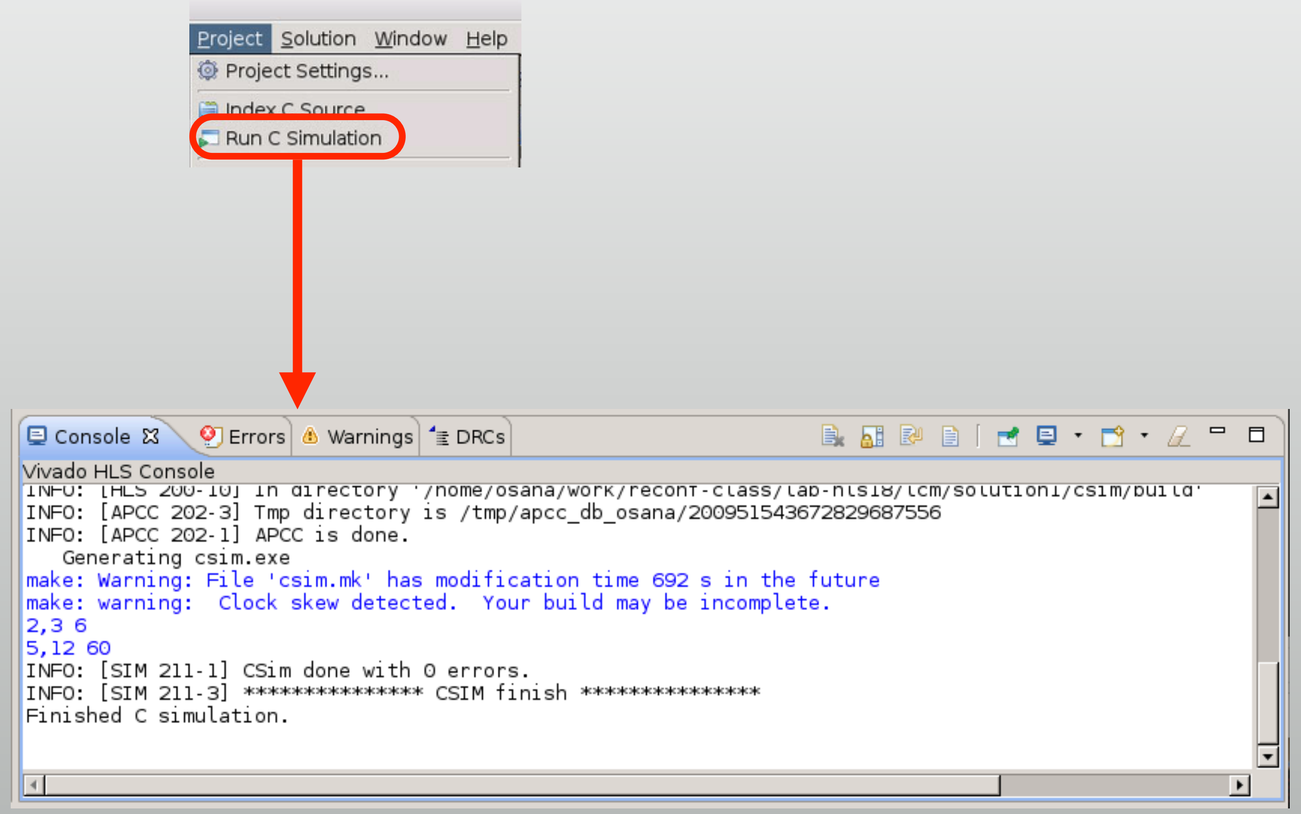


What's the "Solution?"

- 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!

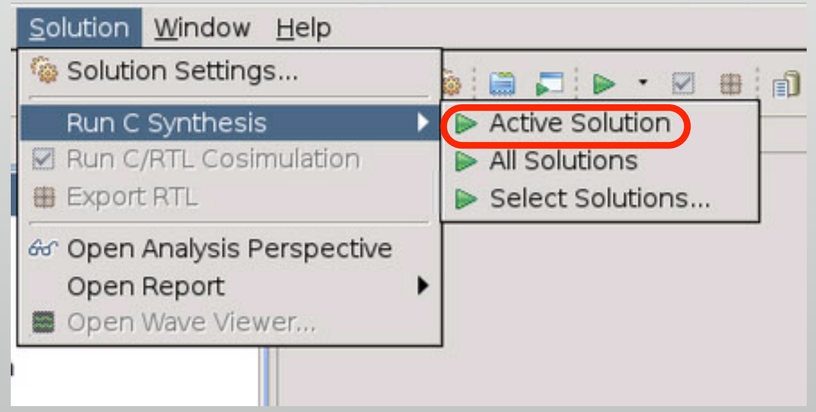
C simulation

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



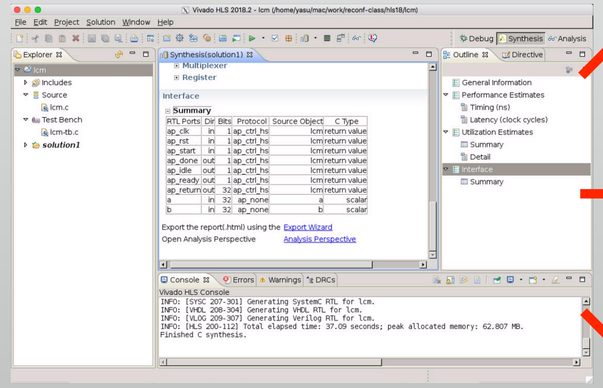
C synthesis

* 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

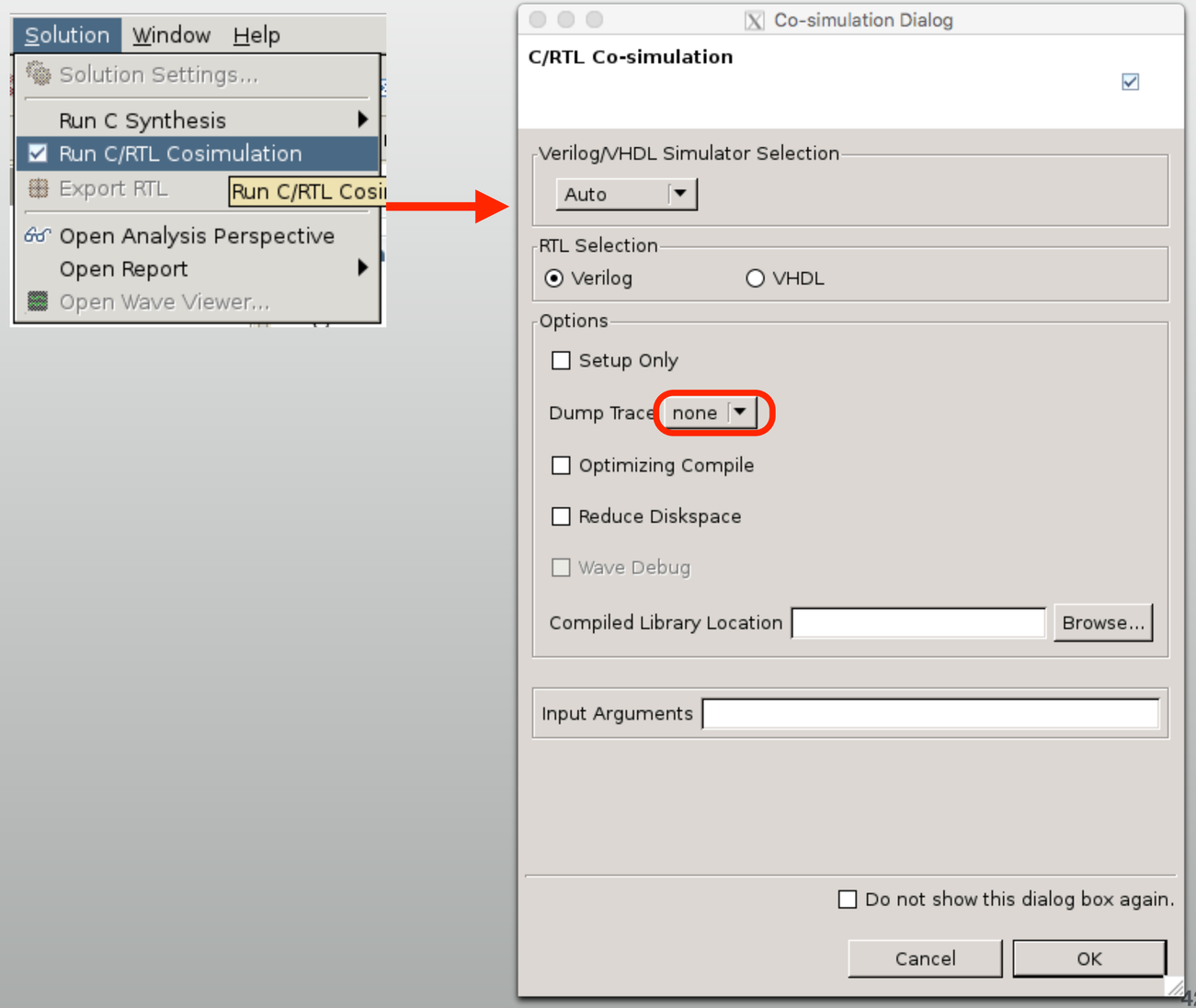
Reading the report

- 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;
```

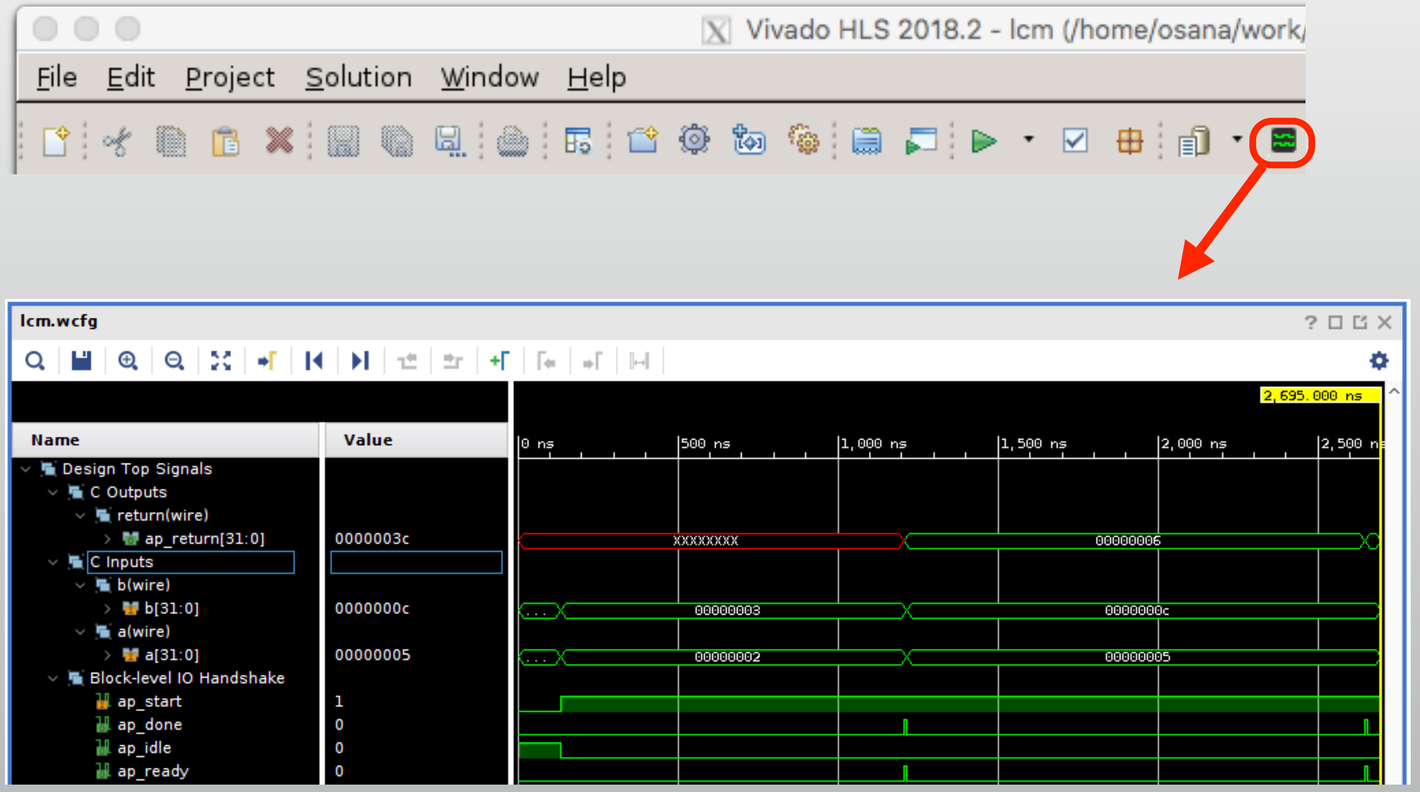
Launch Co-simulation

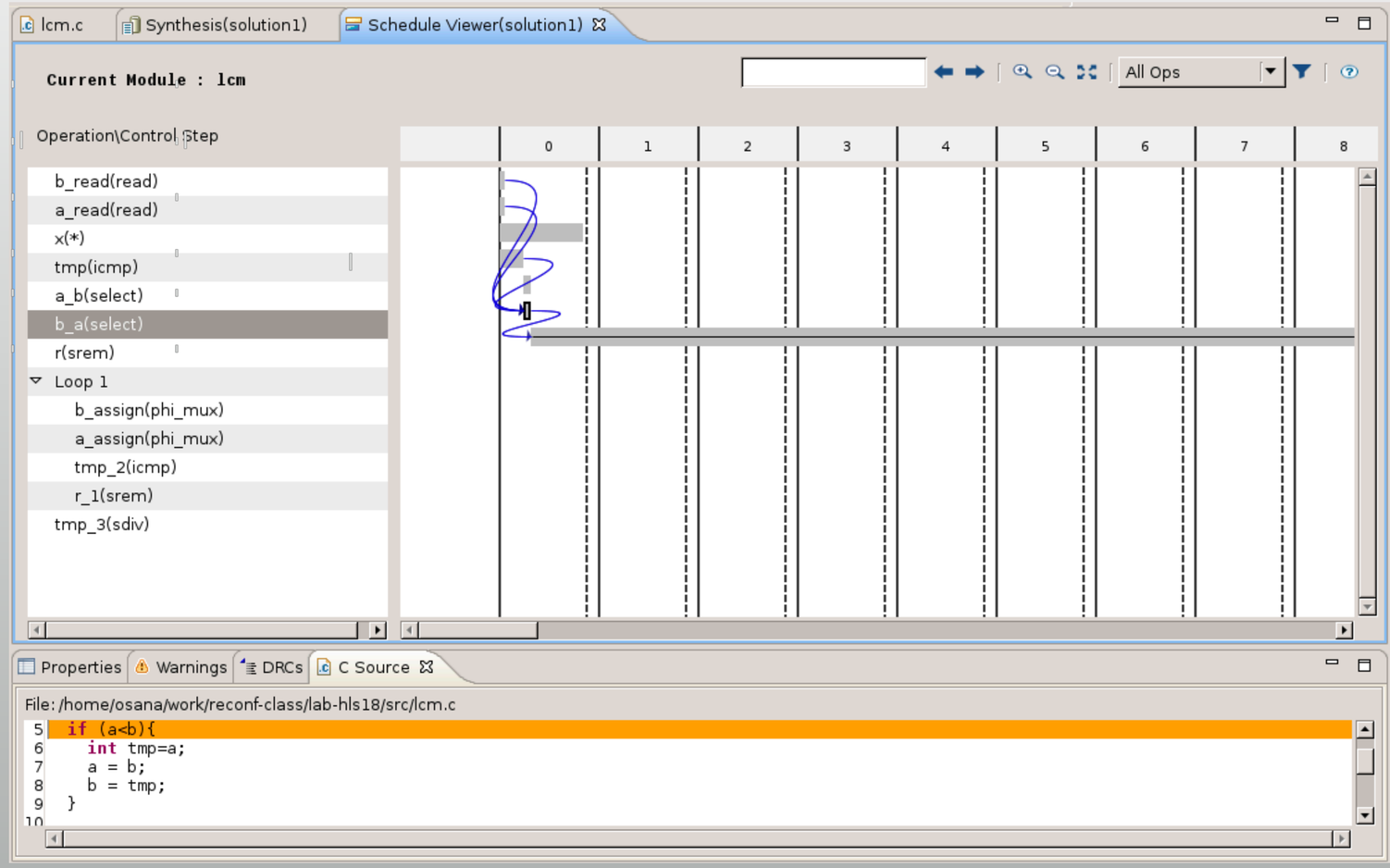
- C TB + Synthesized RTL
  - Set “Dump trace” to “All”
  - C TB is translated to HDL, results are compared



Co-simulation result

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



Analysis View

- Control state and dependency graph
  - Can refer source code