ECE 5775 (Fall'17)
High-Level Digital Design Automation

Vivado HLS Tutorial

Steve Dai, Sean Lai, Zhiru Zhang
School of Electrical and Computer Engineering

Cornell University

CSL

Agenda

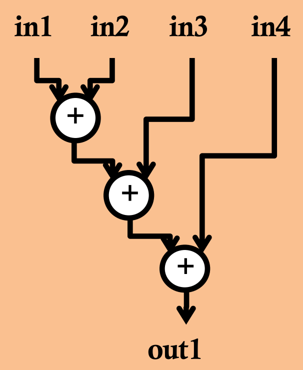
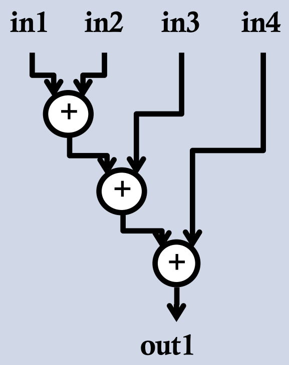
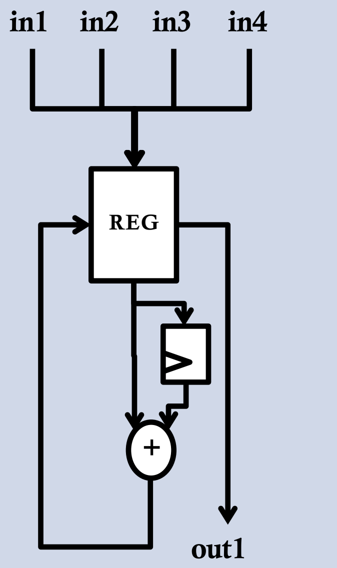
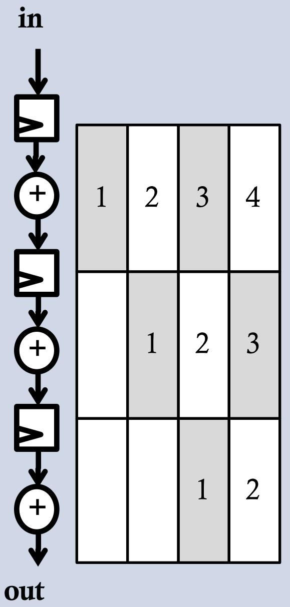
- Logistics and questions
- What is high-level synthesis?
- C-based synthesis
- Case study: FIR filter

High-Level Synthesis (HLS)

- What
  - *Automated* design process that transforms a **high-level functional specification to optimized register-transfer level (RTL)** descriptions for efficient hardware implementation

- Why
  - Productivity
    - lower design complexity and faster simulation speed
  - Permutability
    - rapid design space exploration -> higher quality of result (QoR)
  - Portability
    - single source -> multiple implementations

Permutability: Faster Design Space Exploration

	Latency	Area	Throughput
 **Control-Data Flow Graph**			
$out1 = f(in1, in2, in3, in4)$	$t_{clk} \quad 3 \quad d_{add}$ $T_1 = 1/t_{clk}$ $A_1 = 3 * A_{add}$	$t_{clk} \approx d_{add} + d_{setup}$ $T_2 = 1/(3 * t_{clk})$ $A_2 = A_{add} + 2 * A_{reg}$	$t_{clk} \quad d_{add} + d_{setup}$ $T_3 = 1/t_{clk}$ $A_3 = 3 * A_{add} + 6 * A_{reg}$
Untimed	**Combinational**	**Sequential**	**Pipelined**

Typical C/C++ Synthesizable Subset

- Data types:
  - Primitive types: (u)char, (u)short , (u)int, (u)long, float, double
  - Arbitrary precision integer or fixed-point types
  - Composite types: array, struct, class
  - Templated types: template<>
  - Statically determinable pointers

- No support for dynamic memory allocations

- No support for recursive function calls

Typical C/C++ Constructs to RTL Mapping

C Constructs		HW Components
Functions	→	**Modules**
Arguments	→	**Input/output ports**
Operators	→	**Functional units**
Scalars	→	**Wires or registers**
Arrays	→	**Memories**
Control flows	→	**Control logics**

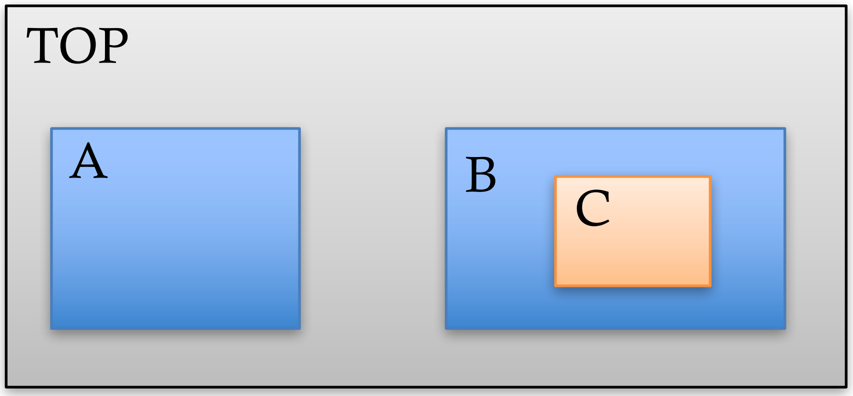
Function Hierarchy

- Each function is usually translated into an RTL module
  - Functions may be inlined to dissolve their hierarchy

Source code

```
void A() { .. body A .. }
void C() { .. body C .. }
void B() {
        C();
}

void TOP( ) {
        A(...);
        B(...);
}
```

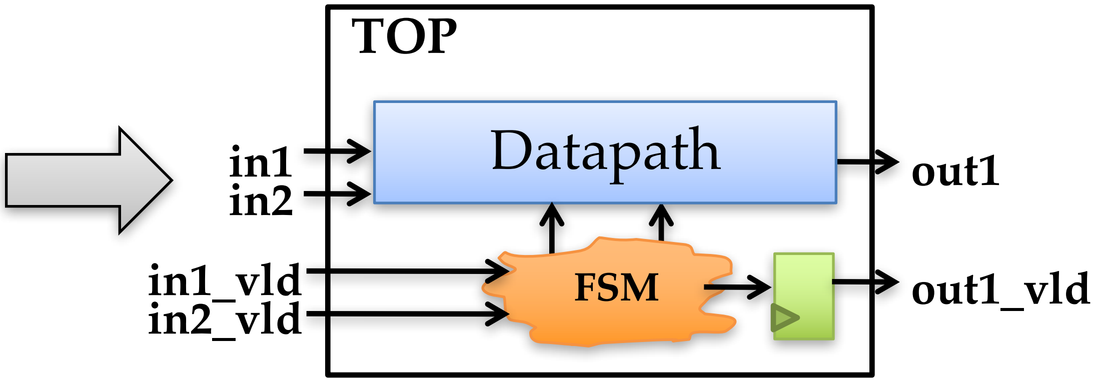
RTL hierarchy



Function Arguments

- Function arguments become ports on the RTL blocks

```
void TOP(int* in1, int* in2,
            int* out1)
{
    *out1 = *in1 + *in2;
}
```



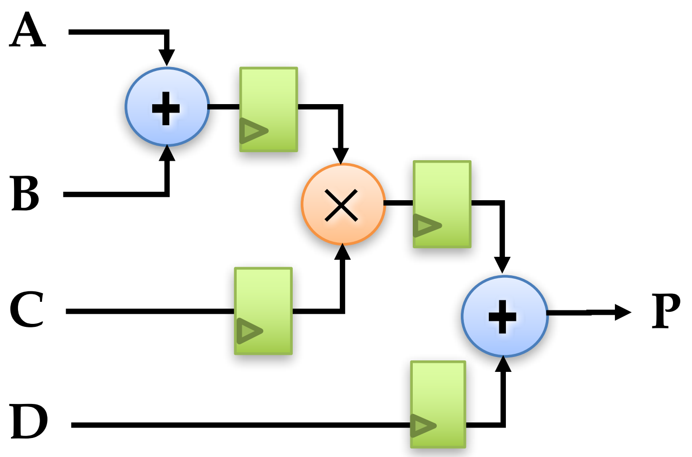
  - Additional control ports are added to the design

- Input/output (I/O) protocols
  - Allow RTL blocks to automatically synchronize data exchange

Expressions

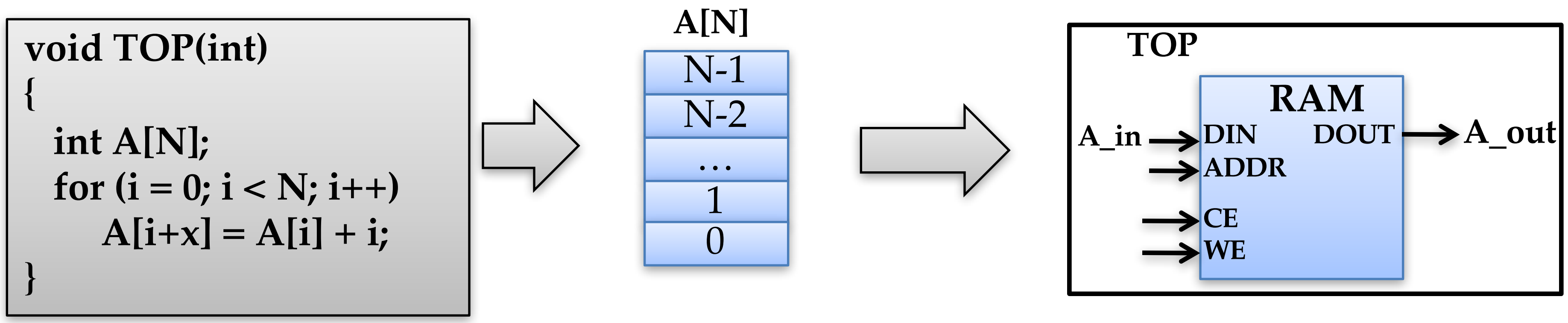
- HLS generates datapath circuits mostly from expressions
  - Timing constraints influence the degree of registering

```
char A, B, C, D,
int P;

P = (A+B)*C+D
```



Arrays

- By default, an array in C code is typically implemented by a memory block in the RTL
  - Read & write array -> RAM; Constant array -> ROM



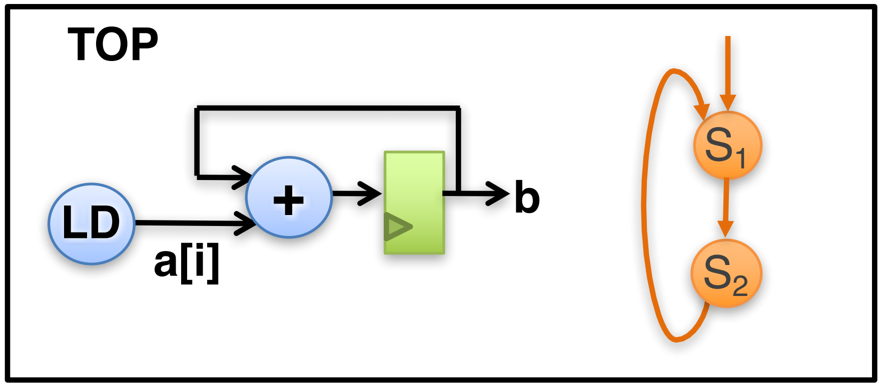
```
void TOP(int)
{
  int A[N];
  for (i = 0; i < N; i++)
     A[i+x] = A[i] + i;
}
```

- An array can be partitioned and map to multiple RAMs
- Multiples arrays can be merged and map to one RAM
- An array can be partitioned into individual elements and map to registers

Loops

- By default, loops are rolled
  - Each loop iteration corresponds to a "sequence" of states (possibly a DAG)
  - This state sequence will be repeated multiple times based on the loop trip count

```
void TOP (...) {
  ...
  for (i = 0; i < N; i++)
       b += a[i];
}
```



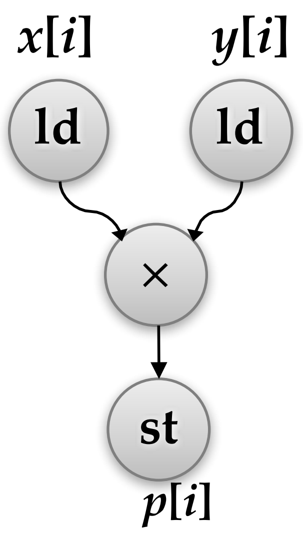
Loop Unrolling

- Loop unrolling to expose higher parallelism and achieve shorter latency
  - Pros
    - Decrease loop overhead
    - Increase parallelism for scheduling
    - Facilitate constant propagation and array-to-scalar promotion
  - Cons
    - Increase operation count, which may negatively impact area, power, and timing

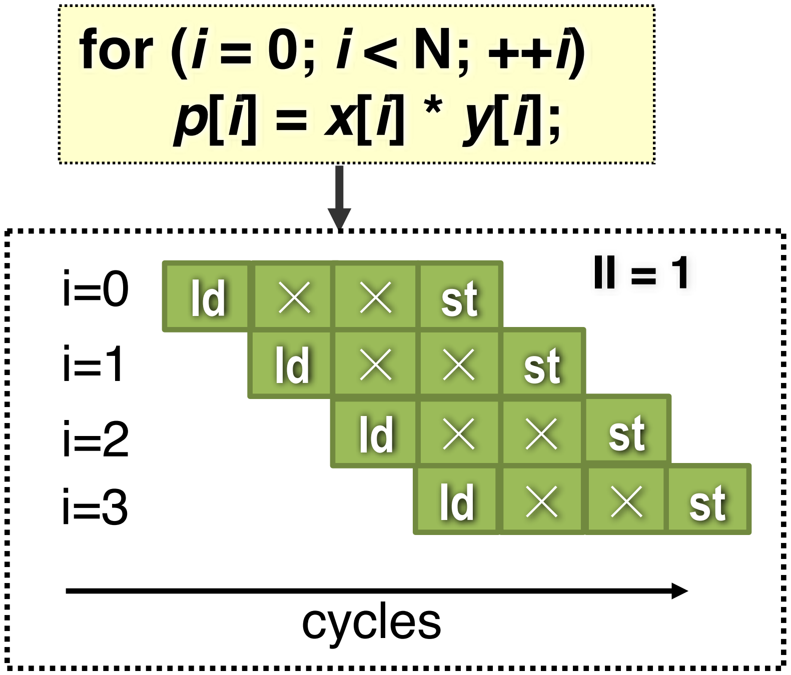
```
for (int i = 0; i < N; i++)
  A[i] = C[i] + D[i];

        ↓

A[0] = C[0] + D[0];
A[1] = C[1] + D[1];
A[2] = C[2] + D[2];
   .....
```

Loop Pipelining

- Loop pipelining is one of the most important optimizations for high-level synthesis
  - Allows a new iteration to begin processing before the previous iteration is complete
  - Key metric: **Initiation Interval (II)** in # cycles



ld – Load
st – Store



for (*i* = 0; *i* < N; ++*i*)
***p*[*i*] = *x*[*i*] * *y*[*i*];**
II = 1
i=0
i=1
i=2
i=3
ld
×
st
cycles

Case Study:
Finite Impulse Response (FIR) Filter

Finite Impulse Response (FIR) Filter

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$

$x[n]$	input signal
$y[n]$	output signal
N	filter order
b_i	*i*th filter coefficient

```
// original, non-optimized version of FIR

#define SIZE 128
#define N 10

void fir(int input[SIZE], int output[SIZE]) {

 // FIR coefficients
 int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};

 // exact translation from FIR formula above
 for (int n = 0; n < SIZE; n++) {
  int acc = 0;
  for (int i = 0; i < N; i++ ) {
   if (n - i >= 0)
     acc += coeff[i] * input[n - i];
  }
  output[n] = acc;
 }
}
```

Server Setup

- Log into ece-linux server
  - Host name: ecelinux.ece.cornell.edu
  - User name and password: [Your NetID credentials]
- Setup tools for this class
  - Source class setup script to setup Vivado HLS

```
> source /classes/ece5775/setup-ece5775.sh
```

- Test Vivado HLS
  - Open Vivado HLS interactive environment

```
> vivado_hls -i
```

  - List the available commands

```
> help
```

Copy FIR Example to Your Home Directory

```
> cd ~
> cp -r /classes/ece5775/FIR_tutorial/ .
> ls
```

- Design files
  - fir.h: function prototypes
  - fir_*.c: function definitions
- Testbench files
  - fir-top.c: function used to test the design
- Synthesis configuration files
  - run.tcl: script for configuring and running Vivado HLS

Project Tcl Script

```
#==================================
# run.tcl for FIR
#==================================

# open the HLS project fir.prj
open_project fir.prj -reset

# set the top-level function of the design to be fir
set_top fir

# add design and testbench files
add_files fir_initial.c
add_files -tb fir-top.c

open_solution "solution1"

# use Zynq device
set_part xc7z020clg484-1

# target clock period is 10 ns
create_clock -period 10
```

```
# do a c simulation
csim_design

# synthesize the design
csynth_design

# do a co-simulation
cosim_design

# close project and quit
close_project

# exit Vivado HLS
quit
```

You can use multiple Tcl scripts to automate different runs with different configurations.

Synthesize and Simulate the Design

```
> vivado_hls -f run.tcl
```

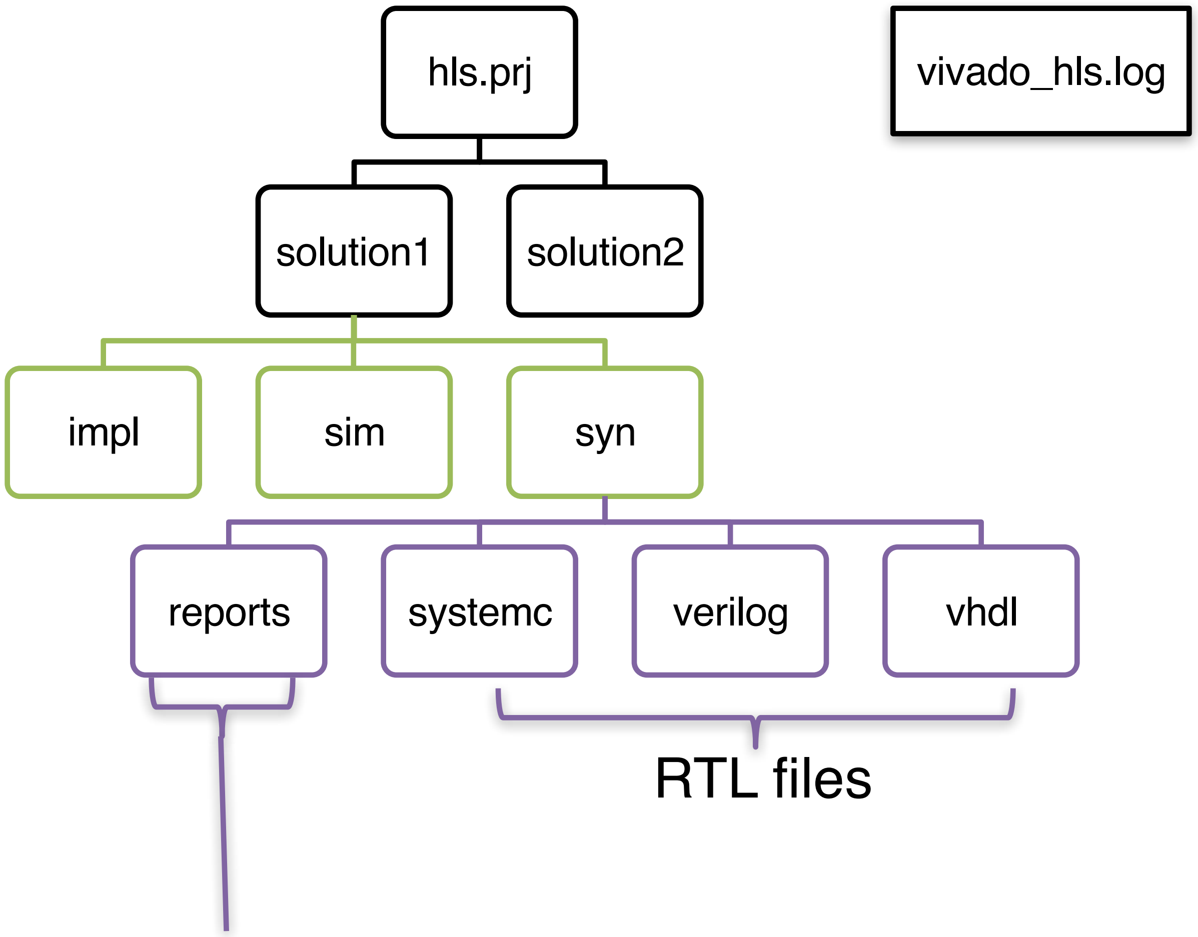
```
Generating csim.exe
128/128 correct values!
INFO: [SIM 211-1] CSim done with 0 errors.

INFO: [HLS 200-10] ----------------------------------------------------------------
INFO: [HLS 200-10] -- Scheduling module 'fir'
INFO: [HLS 200-10] ----------------------------------------------------------------

INFO: [HLS 200-10] ----------------------------------------------------------------
INFO: [HLS 200-10] -- Exploring micro-architecture for module 'fir'
INFO: [HLS 200-10] ----------------------------------------------------------------

INFO: [HLS 200-10] ----------------------------------------------------------------
INFO: [HLS 200-10] -- Generating RTL for module 'fir'
INFO: [HLS 200-10] ----------------------------------------------------------------

INFO: [COSIM 212-47] Using XSIM for RTL simulation.
INFO: [COSIM 212-14] Instrumenting C test bench ...

INFO: [COSIM 212-12] Generating RTL test bench ...
INFO: [COSIM 212-323] Starting verilog simulation.
INFO: [COSIM 212-15] Starting XSIM ...

INFO: [COSIM 212-316] Starting C post checking ...
128/128 correct values!

INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

SW simulation only.
Same as simply running a software program.

HLS
Synthesizing C to RTL

HW-SW co-simulation.
SW test bench invokes RTL simulation.

Synthesis Directory Structure

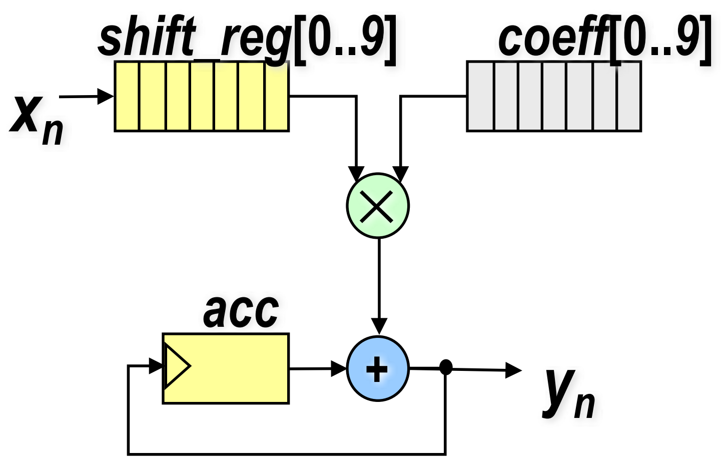


Synthesis reports of each function
in the design, except those inlined.

Default Microarchitecture

```
void fir(int input[SIZE], int output[SIZE]) {
 // FIR coefficients
 int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};
 // Shift registers
 int shift_reg[N] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
 // loop through each output
 for (int i = 0; i < SIZE; i ++ ) {
  int acc = 0;
  // shift registers
  for (int j = N - 1; j > 0; j--) {
   shift_reg[j] = shift_reg[j - 1];
  }
  // put the new input value into the first register
  shift_reg[0] = input[i];
  // do multiply-accumulate operation
  for (j = 0; j < N; j++) {
   acc += shift_reg[j] * coeff[j];
  }
  output[i] = acc;
 }
}
```

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$



Possible optimizations

- Loop unrolling
- Array partitioning
- Pipelining

Unroll Loops

```
void fir(int input[SIZE], int output[SIZE]) {
  …

  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    int acc = 0;
    // shift the registers
    for (int j = N - 1; j > 0; j--) {
      #pragma HLS unroll
      shift_reg[j] = shift_reg[j - 1];
    }
    …
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += shift_reg[j] * coeff[j];
    }
    …
  }
}
```

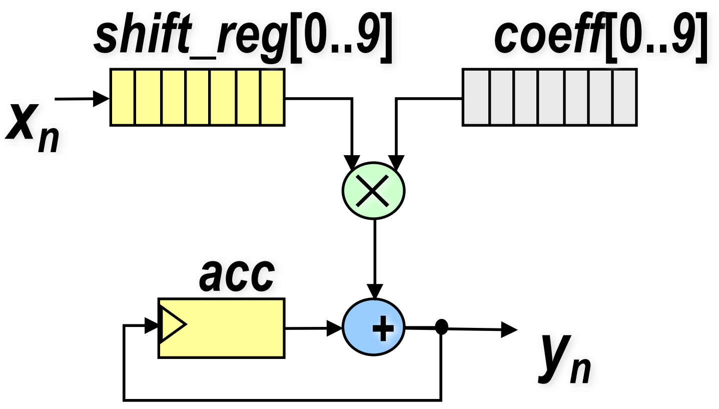
Complete unrolling

```
// unrolled shift registers
shift_reg[9] = shift_reg[8];
shift_reg[8] = shift_reg[7];
shift_reg[7] = shift_reg[6];
…
shift_reg[1] = shift_reg[0];
```

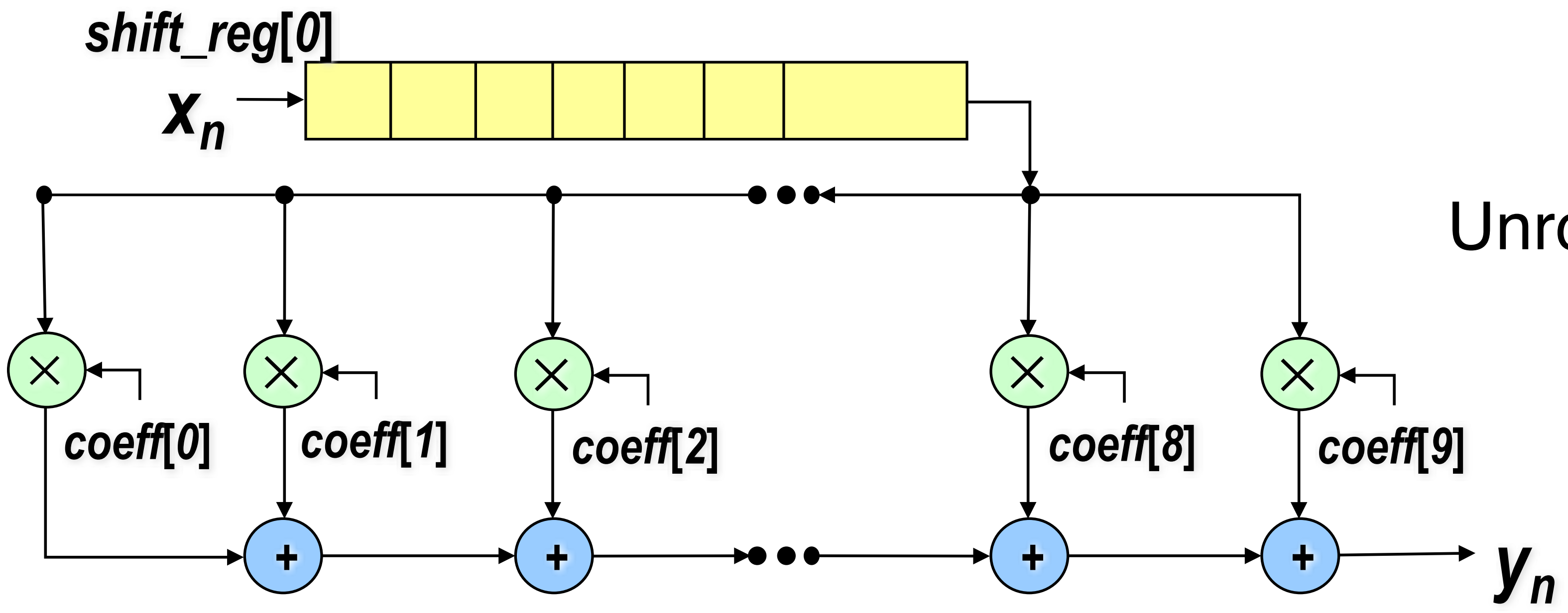
Complete unrolling

```
// unrolled multiply-accumulate
acc += shift_reg[0] * coeff[0];
acc += shift_reg[1] * coeff[1];
acc += shift_reg[2] * coeff[2];
…
acc += shift_reg[9] * coeff[9];
```

Microarchitecture after Unrolling



Default



Unrolled

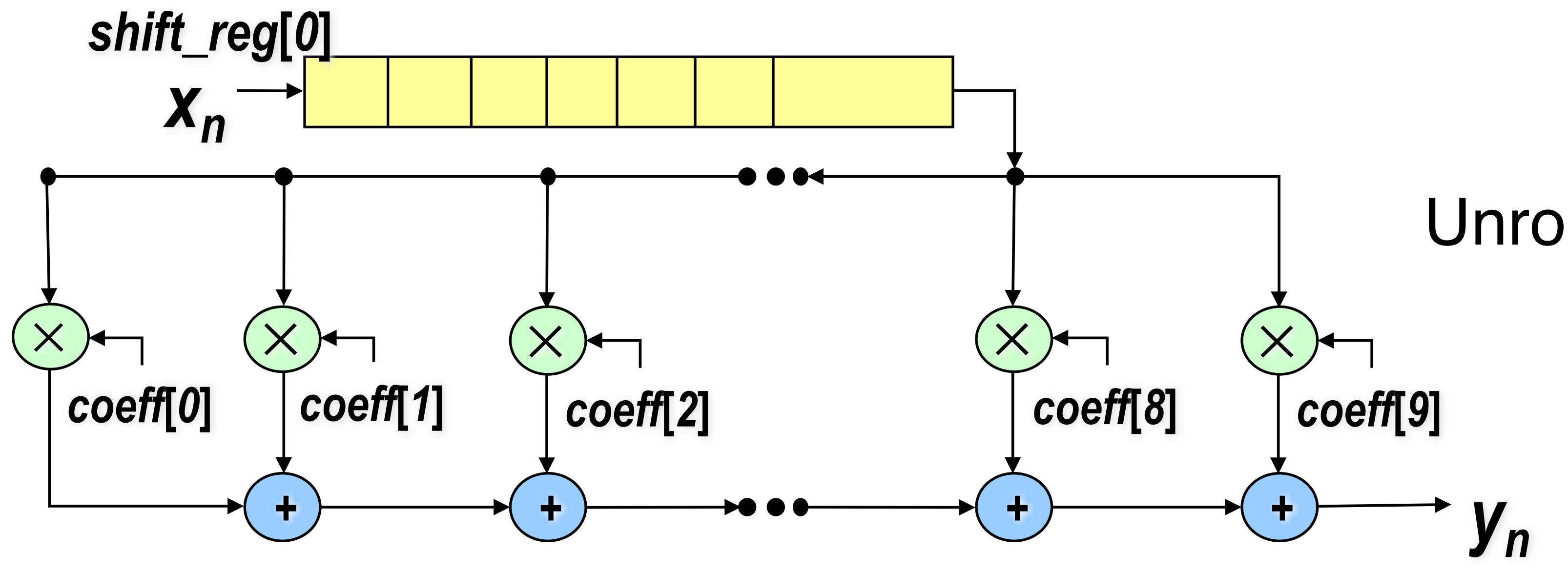
Partition Arrays

```
void fir(int input[SIZE], int output[SIZE]) {
  // FIR coefficients
  int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};
  // Shift registers
  int shift_reg[N] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  #pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=0
  …
}
```

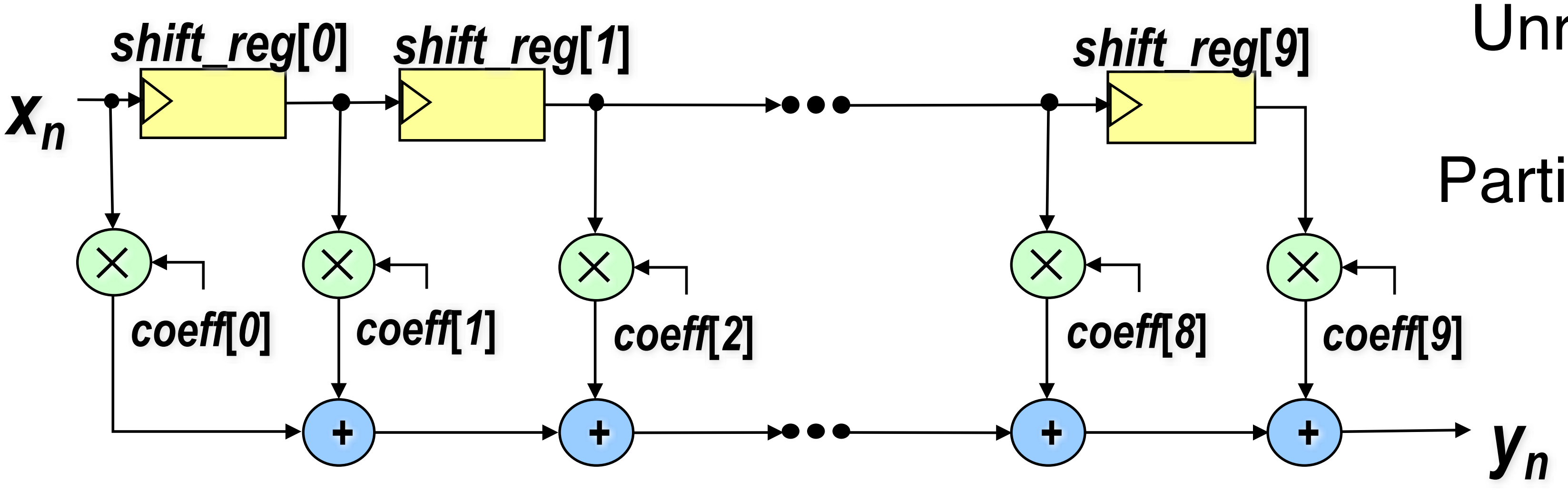
Complete array partitioning

```
// Shift registers
int shift_reg_0 = 0;
int shift_reg_1 = 0;
int shift_reg_2 = 0;
…
int shift_reg_9 = 0;
```

Microarchitecture after Partitioning



Unrolled



Unrolled
+
Partitioned

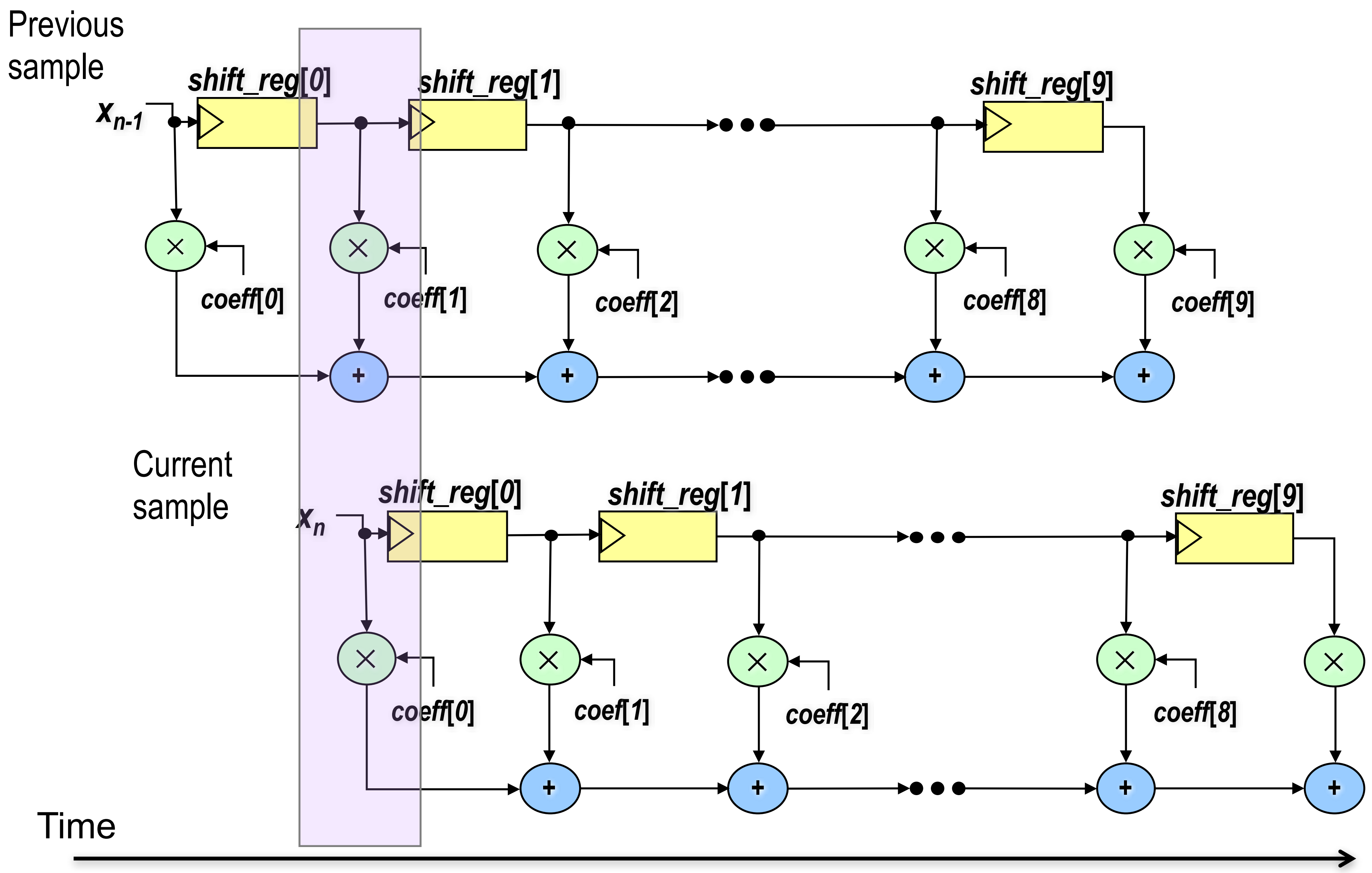
Pipeline Outer Loop

```
void fir(int input[SIZE], int output[SIZE]) {
  …

  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    #pragma HLS pipeline II=1
    int acc = 0;
    // shift the registers
    for (int j = N - 1; j > 0; j--) {
      #pragma HLS unroll
      shift_reg[j] = shift_reg[j - 1];
    }
    …
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += shift_reg[j] * coeff[j];
    }
    …
  }
}
```

Pipeline the entire outer loop

```
// loop through each output
for (int i = 0; i < SIZE; i ++ ) {
  #pragma HLS pipeline II=1
  int acc = 0;

  …

  // put the new input value into the
  // first register
  shift_reg[0] = input[i];

  …
}
```

Fully Pipelined Implementation



Pipeline Outer Loop

```
void fir(int input[SIZE], int output[SIZE]) {
  …

  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    #pragma HLS pipeline II=1
    int acc = 0;
    // shift the registers
    for (int j = N - 1; j > 0; j--) {
      ~~#pragma HLS unroll~~
      shift_reg[j] = shift_reg[j - 1];
    }
    …
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      ~~#pragma HLS unroll~~
      acc += shift_reg[j] * coeff[j];
    }
    …
  }
}
```

Pipeline the entire outer loop

Inner loops automatically unrolled when pipelining the outer loop