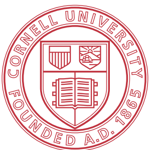
Tutorial on C-Based HLS



Cornell University

Announcements

- Lab 1 will be released today
  - Due: Friday 09/15
- Read this paper before next lecture

Agenda

- Introduction to high-level synthesis (HLS)
  - C-based synthesis
  - Common HLS optimizations
- Matrix-vector multiplication using Vivado HLS
  - Led by Matthew Hofmann (PhD TA)

High-Level Synthesis (HLS)

- What
  - An automated design process that transforms **high-level functional specifications into optimized register-transfer level (RTL)** descriptions for efficient hardware implementation
    - Input spec. to HLS is typically untimed or partially timed

- Why
  - **Productivity**: Lower design complexity & faster simulation speed
  - **Portability**: Single (untimed) source → multiple implementations
  - **Quality:** Quicker design space exploration → higher quality

A Simple Example: RTL vs. HLS

- A GCD unit with handshake

HLS Code

```
void GCD ( msg& req,
           msg& resp ) {
  short a = req.msg_a;
  short b = req.msg_b;
  while ( a != b ) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  resp.msg = a;
}
```

Manual RTL (partial)

```
module GcdUnitRTL
(
  input  wire [  0:0] clk,
  input  wire [ 31:0] req_dat,
  output wire [  0:0] req_rdy,
  input  wire [  0:0] req_val,
  input  wire [  0:0] reset,
  output wire [ 15:0] resp_dat,
  input  wire [  0:0] resp_rdy,
  output wire [  0:0] resp_val
);
```

Module declaration

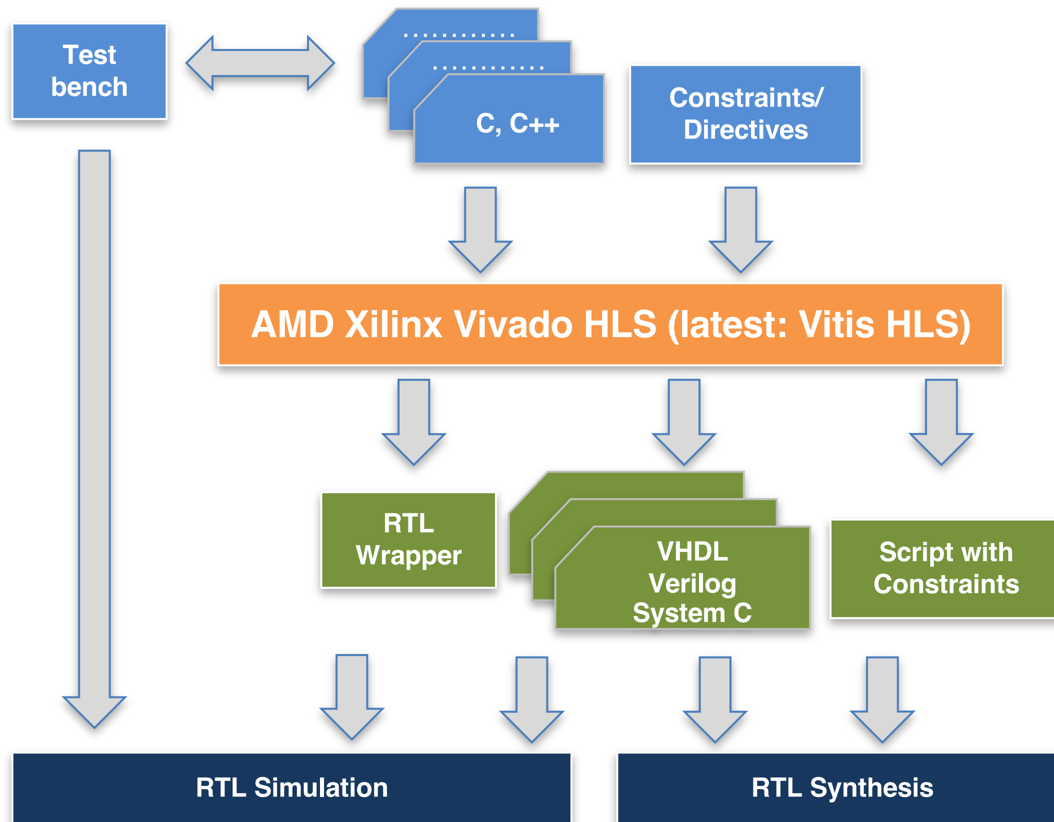
```
always @ (*) begin
  if ((curr_state__0 == STATE_IDLE))
    if (req_val)
      next_state__0 = STATE_CALC;

  if ((curr_state__0 == STATE_CALC))
    if ((!is_a_lt_b&&is_b_zero))
      next_state__0 = STATE_DONE;

  if ((curr_state__0 == STATE_DONE))
    if ((resp_val&&resp_rdy))
      next_state__0 = STATE_IDLE;
end
```

State transition

```
always @ (*) begin
  if ((current_state__1 == STATE_IDLE))
    req_rdy = 1;   resp_val = 0;
    a_mux_sel = A_MUX_SEL_IN;   b_mux_sel = B_MUX_SEL_IN;
    a_reg_en = 1; b_reg_en = 1;

    if ((current_state__1 == STATE_CALC))
      do_swap = is_a_lt_b;   do_sub = ~is_b_zero;
      req_rdy = 0; resp_val = 0;
      a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB;
      a_reg_en = 1; b_reg_en = do_swap;
      b_mux_sel = B_MUX_SEL_A;

    else
      if ((current_state__1 == STATE_DONE))
        req_rdy = 0;   resp_val = 1;
        a_mux_sel = A_MUX_SEL_X;   b_mux_sel = B_MUX_SEL_X;
        a_reg_en = 0;   b_reg_en = 0;
end
```

Output logic

A Representative C-Based HLS Tool



~10X code size reduction with algorithmic specification

~100X simulation speedup

Behavioral-level IP reuse

Fast architecture exploration

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 (or very limited) support for dynamic memory allocations and recursive function calls

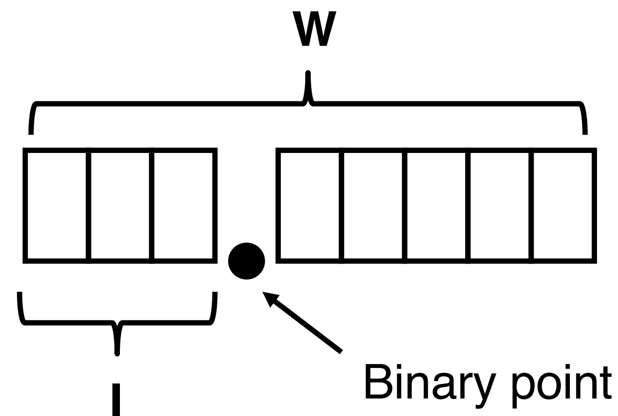
Arbitrary Precision Integer

- C/C++ only provides a limited set of native integer types
  - char (8b), short (16b), int (32b), long (??), long long (64b)
  - Byte aligned: efficient in processors
- Arbitrary precision integer in Vivado HLS
  - Signed: **ap_int**; Unsigned **ap_uint**
    - Two's complement representation for signed integer
  - Templatized class ap_int<W> or ap_uint<W>
    - W is the user-specified bitwidth

```
#include <ap_int>
…
ap_int<9>       x;  // 9-bit
ap_uint<24>   y;  // 24-bit unsigned
ap_uint<512> z;  // 512-bit unsigned
```

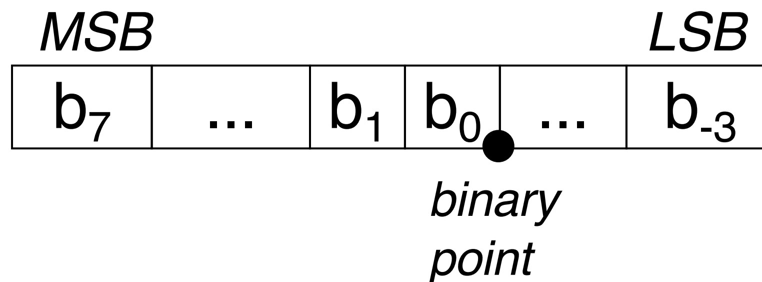
Fixed-Point Type in Vivado HLS

- Arbitrary precision fixed-point type
  - Signed: **ap_fixed**; Unsigned **ap_ufixed**
  - Templatized class ap_fixed<W, I, Q, O>
    - W: total bitwidth
    - I: integer bitwidth
    - Q: quantization mode
    - O: overflow mode



Example: Fixed-Point Modeling

ap_ufixed<11, 8, AP_TRN, AP_WRAP> x;



11 is the total number of bits in the type

8 bits to the left of the decimal point

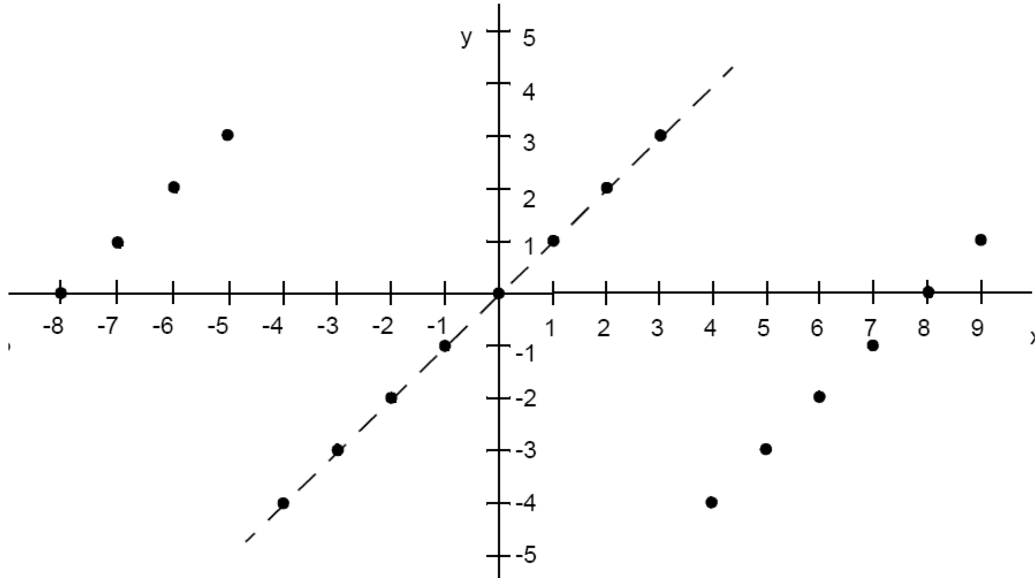
AP_WRAP defines *wrapping* behavior for *overflow*

AP_TRN defines *truncation* behavior for *quantization*

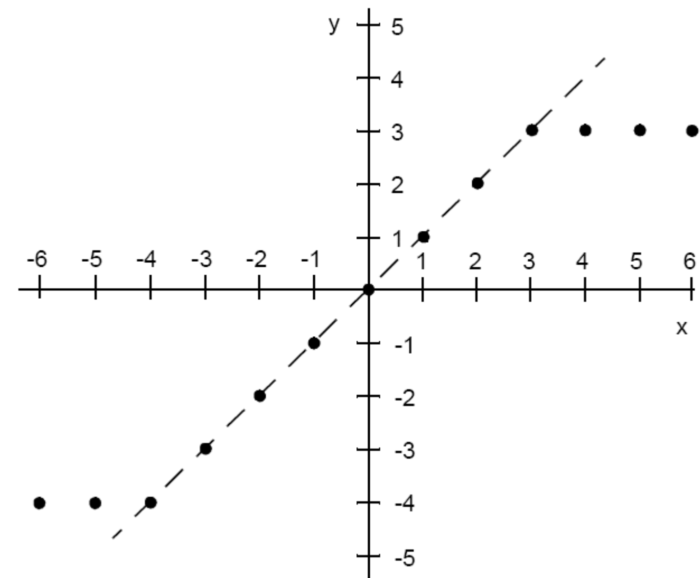
Fixed-Point Type: Overflow Behavior

- ap_fixed overflow mode
  - Determines the behavior of the fixed-point type when the result of an operation generates more precision in the **MSBs** than is available

```
ap_fixed<W, IW_X> x;
ap_fixed<W, IW_Y> y = x; // assuming IW_Y < IW_X
```



Default: AP_WRAP (wrapping mode)



AP_SAT (saturation mode)

Fixed-Point Type: Quantization Behavior

- ap_fixed quantization mode
  - Determines the behavior of the fixed-point type when the result of an operation generates more precision in the LSBs than is available
  - Default mode: AP_TRN (truncation)
  - Other rounding modes: AP_RND, AP_RND_ZERO, AP_RND_INF, …

```
ap_fixed<4, 2, AP_TRN>   x = 1.25;    (b'01.01)
ap_fixed<3, 2, AP_TRN>   y = x;
                         └──> 1.0     (b'01.0)

ap_fixed<4, 2, AP_TRN>   x = -1.25;   (b'10.11)
ap_fixed<3, 2, AP_TRN>   y = x;
                         └──> -1.5    (b'10.1)
```

Typical C/C++ Constructs to RTL Mapping

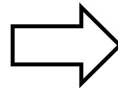
C/C++ Constructs		RTL Components
Functions	→	**Modules**
Arguments	→	**Input/output ports**
Operators	→	**Functional units**
Scalars	→	**Wires or registers**
Arrays	→	**Memories**
Control flows	→	**Control logics**

Functions and Design Hierarchy

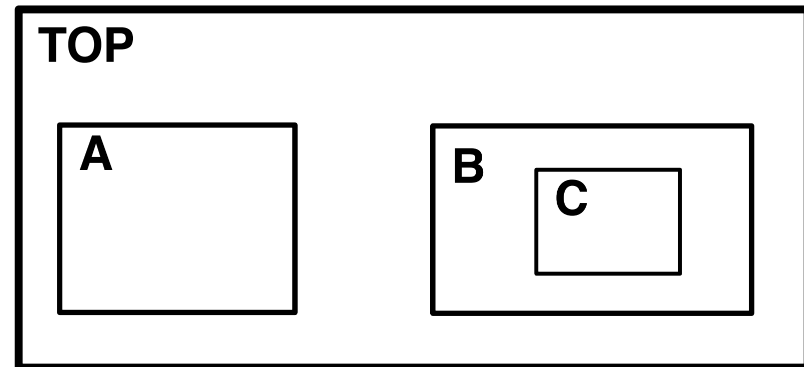
- Each function is usually translated into an RTL module
  - Function arguments become ports on the RTL blocks
  - Functions may be inlined to dissolve their hierarchy

Source code

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

void TOP(int x, int* y) {
  A( ); B( );
}
```

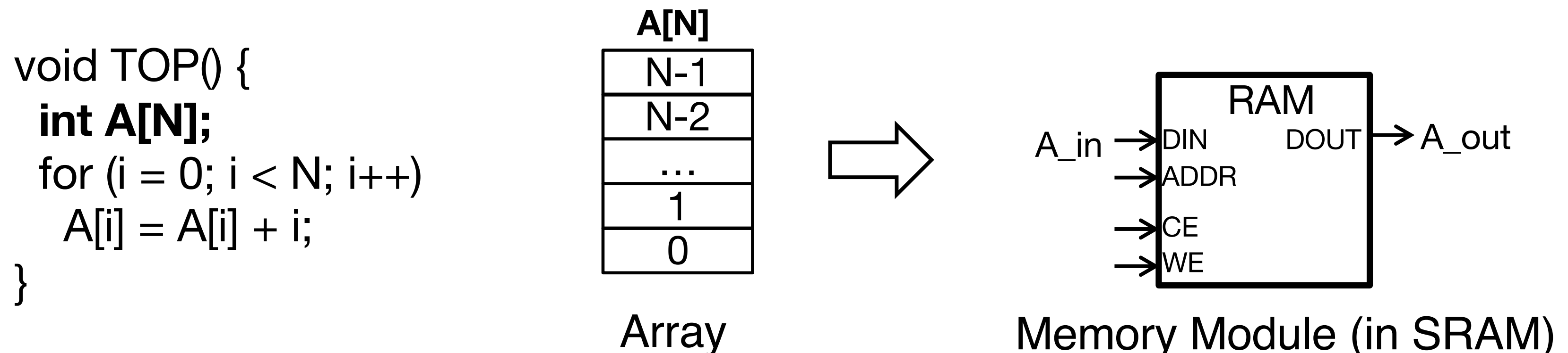


RTL module hierarchy



Arrays

- An array is usually implemented by a memory module in RTL
  - Reading and writing to the array correspond to accessing RAM, while constant arrays are stored in ROM
  - Typically, each memory module supports a limited number of read/write ports, typically up to 2

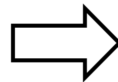


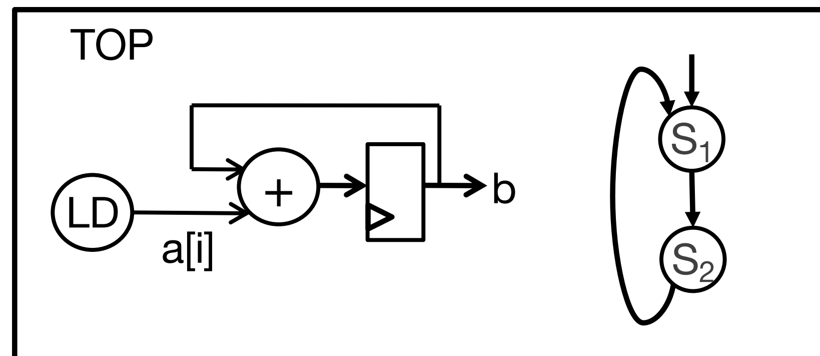
- An array can be partitioned and implemented with multiple RAMs
  - Extreme case: completely partitioned into individual elements that map to discrete registers
- Multiples arrays can be merged and mapped to one RAM

Loops

- By default, loops are "rolled"
  - Each loop iteration corresponds to a "sequence" of states (more generally, an FSM)
  - This state sequence will be repeated multiple times based on the loop trip count (or loop bound)

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



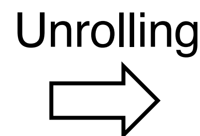


Loop Unrolling

- Unrolling can expose more parallelism to achieve shorter latency or higher throughput
  - (+) Decreased loop control overhead
  - (+) Increased parallelism for scheduling
  - (–) Increased operation count, which may negatively impact area, timing, and power

```
for (int i = 0; i < 8; i++)
  A[i] = C[i] + D[i];
```

Unrolling

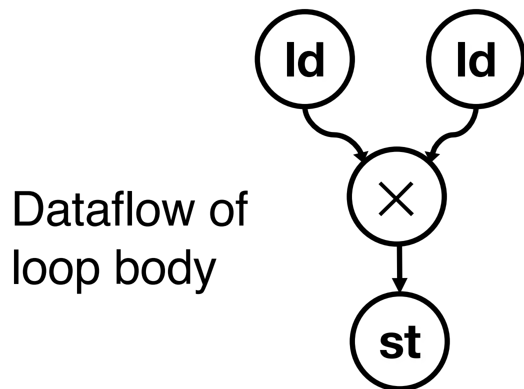


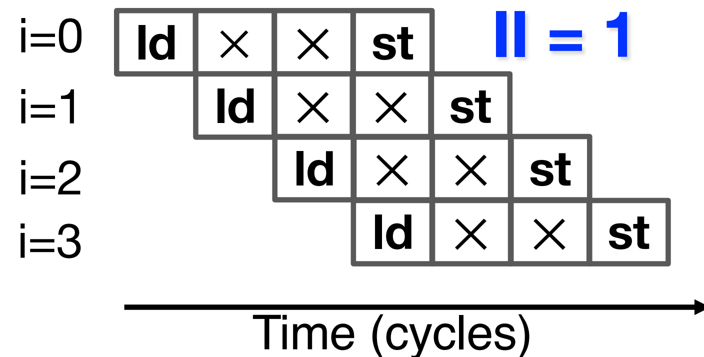
```
A[0] = C[0] + D[0];
A[1] = C[1] + D[1];
A[2] = C[2] + D[2];
    ...
A[7] = C[7] + D[7];
```

Loop Pipelining

- Pipelining is one of the most important optimizations for HLS
  - Key factor: **Initiation Interval (II)**
  - Allows a new iteration to begin processing, II cycles after the start of the previous iteration **(II=1 means the loop is fully pipelined)**

```
for (i = 0; i < N; ++i)
    p[i] = x[i] * y[i];
```

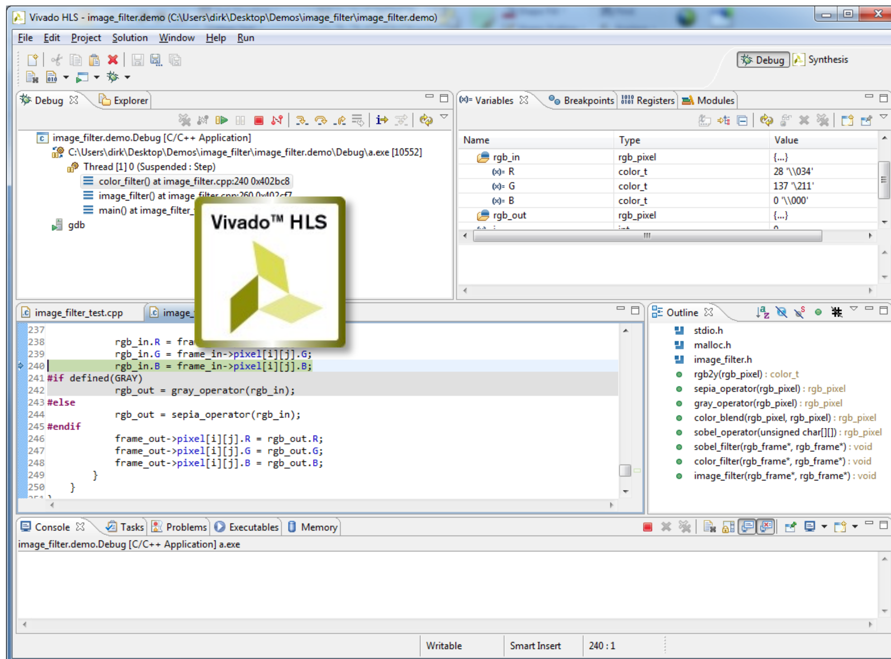


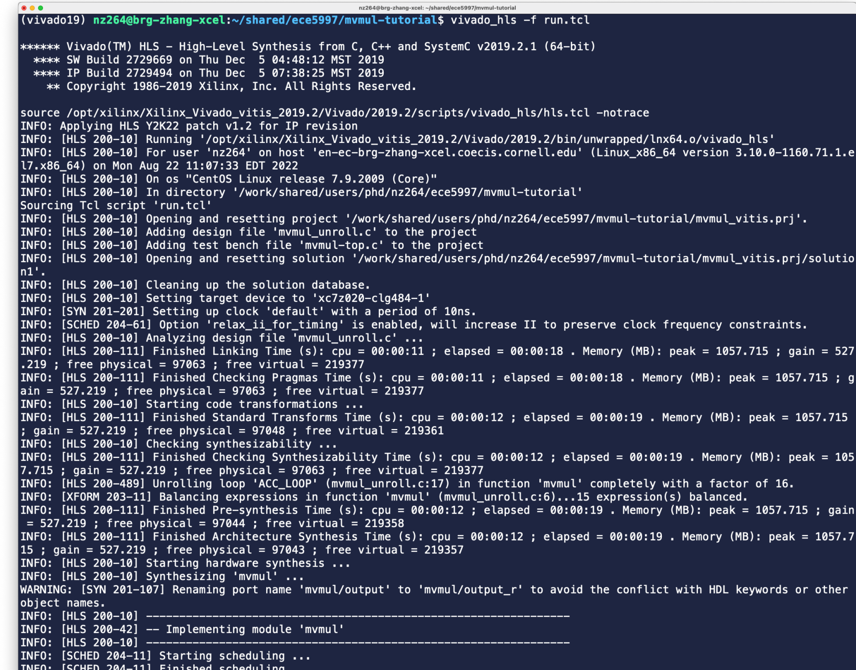


ld – Load (memory read)
st – Store (memory write)

Here we assume multiplication (×) takes two cycles

A Tutorial on Vivado HLS





AMD Xilinx Vivado HLS (v2019.2)

(We will exclusively use the command-line interface)

Matrix-Vector Multiplication

$$\mathbf{y} = A\,\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N \\ \vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N \end{bmatrix}$$

A: input matrix

x : input vector

y : output vector

```
// original, non-optimized version of MV
#define N 16 // dimension of
matrix/vector

// y = Mx
void MV ( int A[N][N], int x[N], int y[N] ) {
  // iterate over rows of matrix A
  for ( int i = 0; i < N; i++ ) {
    int acc = 0;
    // inner product
    for ( int j = 0; j < N; j++ ) {
        acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

Setup on ECE Linux Server

- Log into ecelinux server

```
> ssh <netid>@ecelinux.ece.cornell.edu
```

  - More info: it.coecis.cornell.edu/ece/ecelinux/
- Get Vivado HLS tool in your environment
  - Source class setup script to setup Vivado HLS

```
> source /classes/ece6775/setup-ece6775.sh
```

- Test Vivado HLS
  - Open Vivado HLS in interactive mode

```
> vivado_hls -i
```

  - List the available commands; then quit

```
> help
```

```
> quit
```

Copy MV Example to Your Home Directory

```
> cd ~
> cp -r /classes/ece6775/mv-tutorial/ .
> ls
```

- Design files
  - `mv.h`: header file declaring top-level function
  - `mv_*.cpp`: HLS function definitions
- Testbench files
  - `testbench.cpp`: code for testing correctness
- Synthesis Tcl (tickle) Script
  - `run.tcl`: script for configuring and running Vivado HLS

Example Tcl Script

```
#====================================
# run.tcl for matrix-vector (MV) multiplication
#====================================

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

# set the top-level function of the design to mv
set_top mv

# add design and testbench files
add_files mv_initial.cpp
add_files -tb testbench.cpp

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

# export design
export_design

# exit Vivado HLS
exit
```

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

Synthesize and Simulate the Design

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

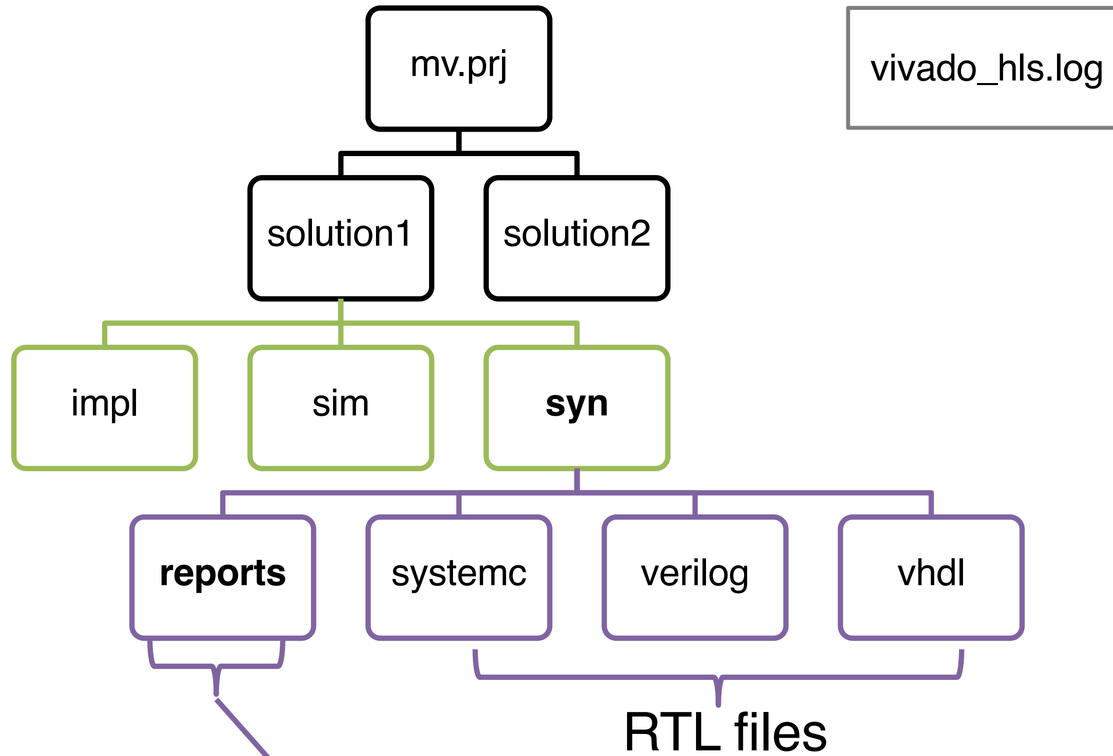
```
Generating csim.exe
128/128 correct values!
INFO: [SIM 211-1] CSim done with 0 errors.

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

INFO: [HLS 200-10] ----------------------------------------------------------------
INFO: [HLS 200-10] -- Exploring micro-arch for module 'mv'
INFO: [HLS 200-10] ----------------------------------------------------------------

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

Synthesis
Compiling C to RTL

HW-SW co-simulation
RTL simulation driven by SW test bench

Synthesis Directory Structure

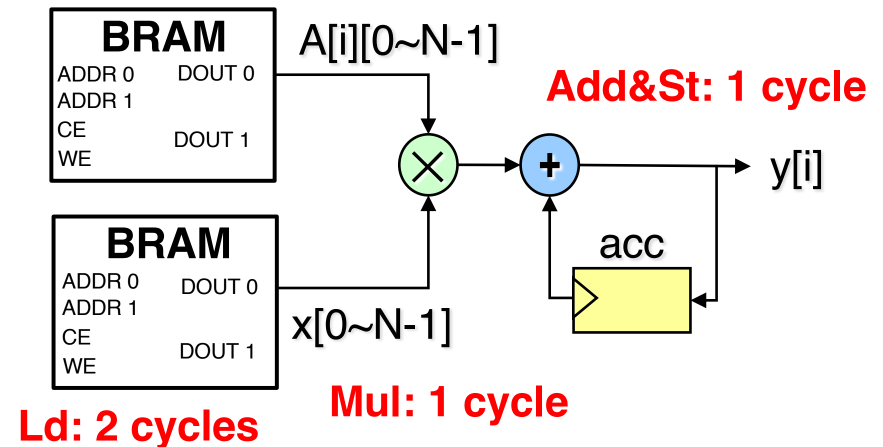


Synthesis reports of each function
in the design, except those inlined.

Default Microarchitecture

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

#define N 16 // dimension of matrix/vector

void MV ( int A[N][N], int x[N], int y[N] ) {
  // iterate over rows of matrix A
  for ( int i = 0; i < N; i++ ) {
    int acc = 0;
    // inner product
    for ( int j = 0; j < N; j++ ) {
        acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

$$y[i] = \sum_{j=0}^{N} A[i][j] \times x[j]$$



Assumption: A and x are stored in two discrete **block RAMs (BRAMs)**; each has 2 read/write ports

Latency: ??

Examine Overall Latency

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

#define N 16 // dimension of matrix/vector

void MV ( int A[N][N], int x[N], int y[N] ) {
  // iterate over rows of matrix A
  for ( int i = 0; i < N; i++ ) {
    int acc = 0;
    // inner product
    for ( int j = 0; j < N; j++ ) {
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

4 cycles (inner loop body)

$4\times16 + 2$ = 66 cycles (outer loop body)

Latency: ~1056 = 66×16

Possible optimizations

- Unrolling
- Pipelining
- Array partitioning

Unroll Inner Loop

```
// MV with inner loop unroll
#define N 16

void MV ( int A[N][N], int x[N], int y[N] ) {
 // iterate over rows of matrix A
 for (int i = 0; i < N; i++) {
   int acc = 0;
    // inner product
   for (int j = 0; j < N; j++ ) {
       #pragma HLS unroll
       acc += A[i][j] * x[j];
   }
   y[i] = acc;
  }
}
```

Complete unrolling

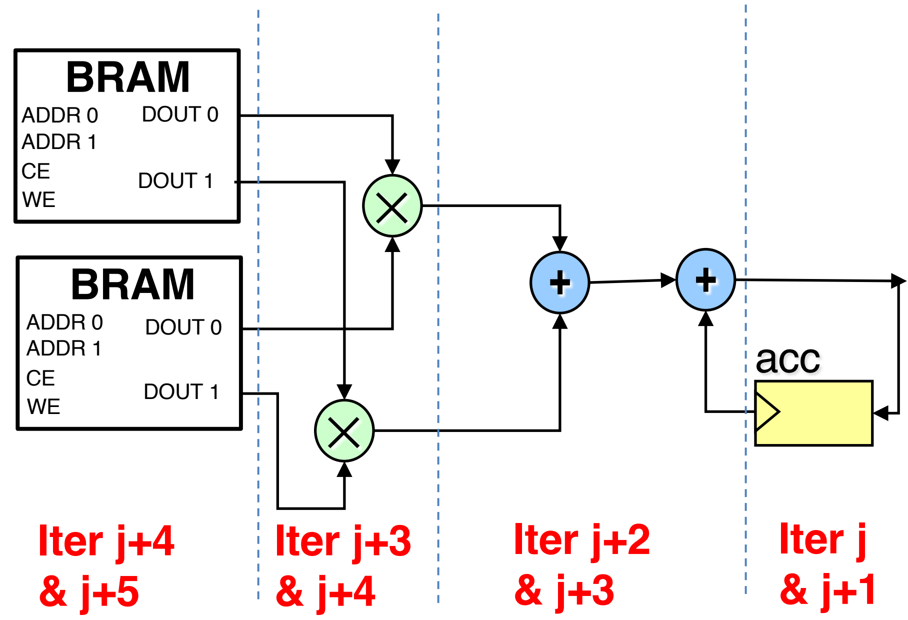
```
// unrolled multiply-accumulate
acc +=  A[i][0] * x[0];
acc +=  A[i][1] * x[1];
acc +=  A[i][2] * x[2];
…
acc +=  A[i][15] * x[15];
```

Latency: ??

Unroll Inner Loop

```
// unrolled multiply-accumulate
acc +=  A[i][0] * x[0];
acc +=  A[i][1] * x[1];
acc +=  A[i][2] * x[2];
…
acc +=  A[i][15] * x[15];
```

Scheduled body



Latency: ??

Overall Latency after Unrolling

```
// MV with inner loop unroll
#define N 16

void MV ( int A[N][N], int x[N], int y[N] ) {
 // iterate over rows of matrix A
 for (int i = 0; i < N; i++) {
    int acc = 0;
    // inner product
    acc +=  A[i][0] * x[0];
    acc +=  A[i][1] * x[1];
    acc +=  A[i][2] * x[2];
    …
    acc +=  A[i][15] * x[15];
    y[i] = acc;
  }
}
```

12 cycles

Default micro-architecture

- A and x are mapped to dual ported RAMs
- We can create 2 instances of the dataflow circuit
- Inner product latency:

  4 + 16/2 = 4 + 8 = 12 cycles

Latency: ??

Pipeline Outer Loop

```
// MV with outer loop pipeline
#define N 16

void MV ( int A[N][N], int x[N], int y[N] ) {
  // iterate over rows of matrix A
  for (int i = 0; i < N; i++) {
    #pragma HLS pipeline
    int acc = 0;
    // inner product
    for (int j = 0; j < N; j++ ) {
        #pragma HLS unroll
        acc += A[i][j] * x[j];
    }
    output[i] = acc;
  }
}
```

(The line `#pragma HLS unroll` is struck through on the slide.)

Pipeline the outer loop

Inner loops automatically unrolled when pipelining the outer loop

II = ??

Overall Latency after Pipelining

```
// MV with outer loop pipeline
#define N 16

void MV ( int A[N][N], int x[N], int y[N] ) {
 // iterate over rows of matrix A
 for (int i = 0; i < N; i++) {
 #pragma HLS pipeline
    int acc = 0;
    // inner product
    acc +=  A[i][0] * x[0];
    acc +=  A[i][1] * x[1];
    acc +=  A[i][2] * x[2];
    …
    acc +=  A[i][15] * x[15];
    y[i] = acc;
  }
}
```

12 cycles

Target II = 1

Achieved II = ??

Why? Pipeline rate limited by BRAM ports

16 reads on A (or x), but only two ports available

Latency: ~132

Partition Arrays

```
// MV with outer loop pipeline and array partition
#define N 16
void MV(int A[N][N], int x[N], int y[N]) {
  #pragma HLS array_partition variable=A dim=2
  #pragma HLS array_partition variable=x
  // iterate over rows of matrix A
  for (int i = 0; i < N; i++) {
   #pragma HLS pipeline
   // inner product
    for (int j = 0; j < N; j++ ) {
     …
  }
```

Latency: ??

```
// matrix col arrays
int A_col_0[16];
int A_col_1[16];
int A_col_2[16];
…
int A_col_15[16];
```

Partition along second axis (column)

```
// vector elements
int x_reg_0;
int x_reg_1;
int x_reg_2;
…
int x_reg_15;
```

Complete array partitioning

Overall Latency after Partitioning

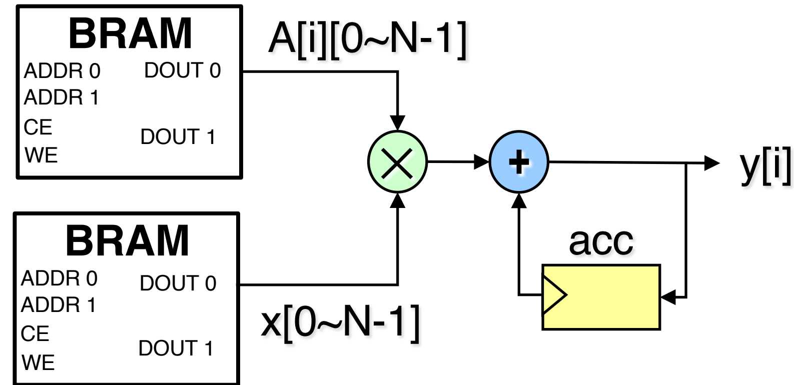
```
// MV with outer loop pipeline
#define N 16

void MV ( int A[N][N], int x[N], int y[N] ) {
 #pragma HLS array_partition variable=A dim=2
 #pragma HLS array_partition variable=x
// iterate over rows of matrix A
 for (int i = 0; i < N; i++) {
 #pragma HLS pipeline
    int acc = 0;
    // inner product
    acc +=  A[i][0] * x[0];
    acc +=  A[i][1] * x[1];
    acc +=  A[i][2] * x[2];
    …
    acc +=  A[i][15] * x[15];
    y[i] = acc;
  }
}
```

6 cycles

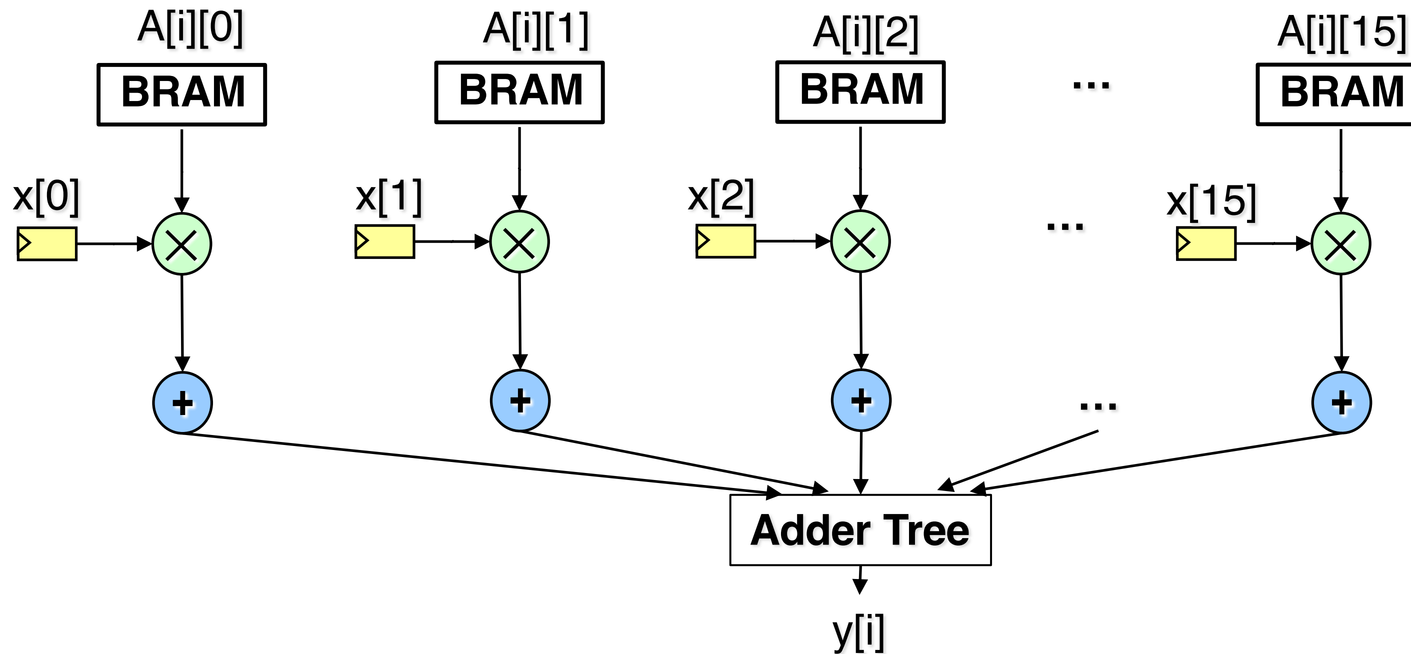
Target II = 1
Achieved II = 1

Latency: ~21 = 6 + (16 – 1) × 1

Microarchitecture of the Optimized Design

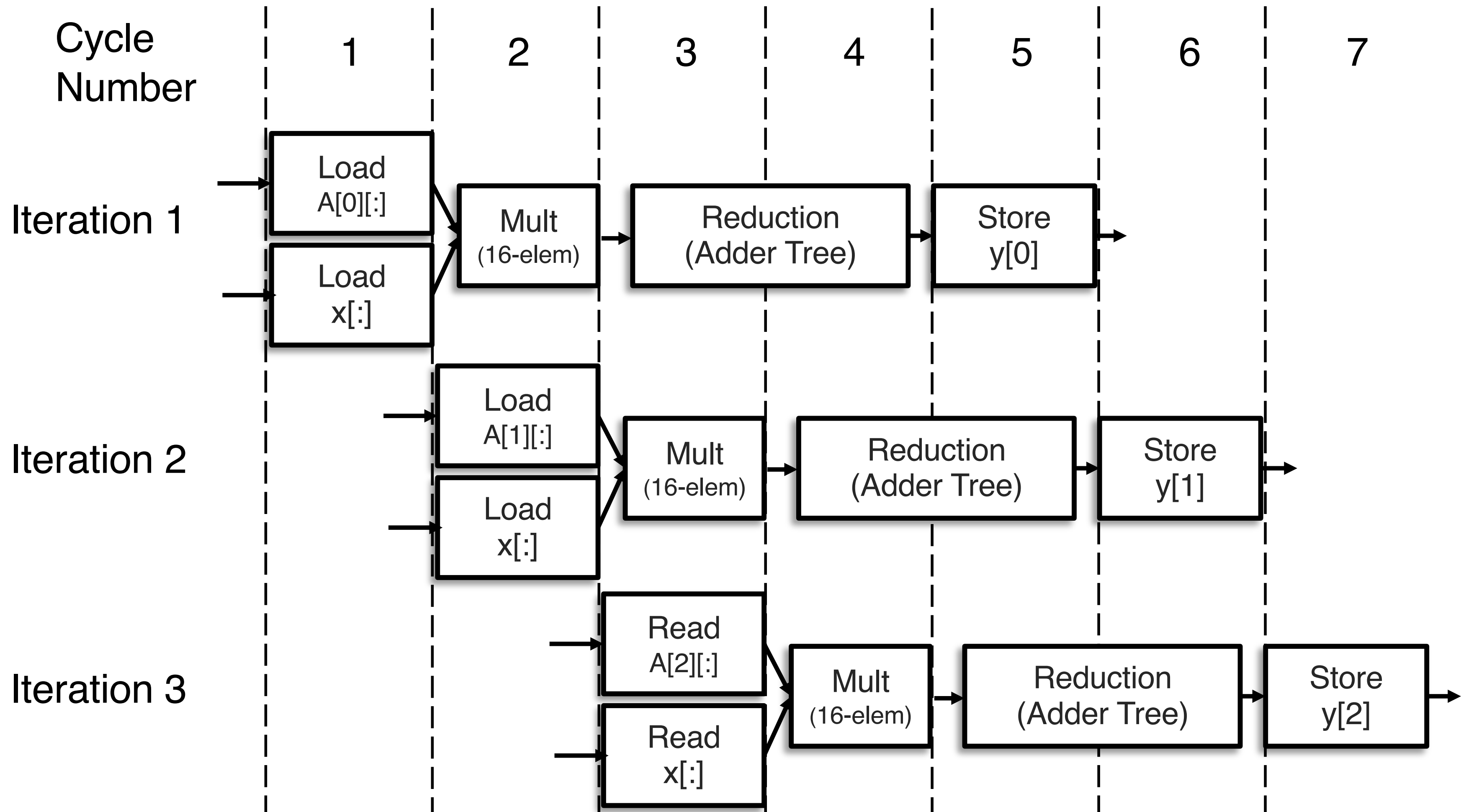


Default



Unrolled
+
Pipelined
+
Partitioned

Pipeline Schedule of the Optimized Design



Next Lecture

- Field-programmable gate arrays (FPGAs)