Pipelining, Xilinx Vitis HLS Data Types and QR Arrays

Joseph Cavallaro
Rice University
20 October 2022

Last Lecture

- Begin Computer Arithmetic for CORDIC
  - ⇒ Sine and Cosine functions in hardware
  - ⇒ Vector or Givens rotations of a vector in hardware
  - ⇒ Useful for communication systems “up converters” and for matrix linear algebra functions

Today

- Vitis HLS Data Types
- Notion of Data Flow and pipeline scheduling – Wolf Chapter 6
- Review of CORDIC and intro to QR
- QR Decomposition array structures

Vitis HLS Data Types

- **C and C++ have standard types created on the 8-bit boundary**
  - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
- **Arbitrary precision in C is supported using apint and ap_int in C++**
  - Compile using apcc for arbitrary precision
  - Arbitrary precision types can define bit-accurate operators leading to better QoR
- **Fixed point precision is supported in C++**
  - Both signed (ap_fixed) and unsigned types (ap_ufixed)

Vitis HLS Data Types

- **Various quantization and overflow modes supported**
  - Quantization
    - AP_RND, AP_RND_ZERO, AP_RND_MIN_INF, AP_RND_INF, AP_RND_CONV, AP_TRN, AP_TRN_ZERO
  - Overflow
    - AP_SAT, AP_SAT_ZERO, AP_SAT_SYM, AP_WRAP, AP_WRAP_SYM
- **Both single- and double-precision floating point data types are supported**
  - If a corresponding floating point core is available then it will automatically be used
  - If floating point core is not available then Vitis HLS will generate the RTL model

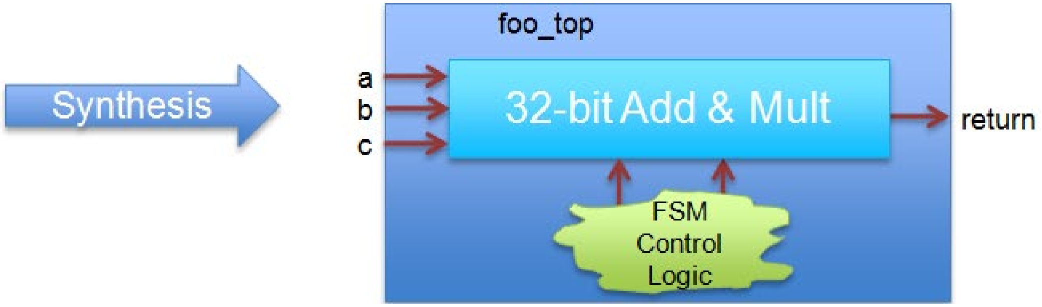
Data Types and Bit-Accuracy

- **C and C++ have standard types created on the 8-bit boundary**
  - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
    - Also provides stdint.h (for C), and stdint.h and cstdint (for C++)
    - Types: int8_t, uint16_t, uint32_t, int_64_t etc.
  - They result in hardware which is not bit-accurate and can give sub-standard QoR
- **Vitis HLS provides bit-accurate types in both C and C++**
  - Allow any arbitrary bit-width to be specified
  - Hence designers can improve the QoR of the hardware by specifying exact data widths
    - Can be specified in the code and simulated to ensure there is no loss of accuracy

Why is arbitrary precision Needed?

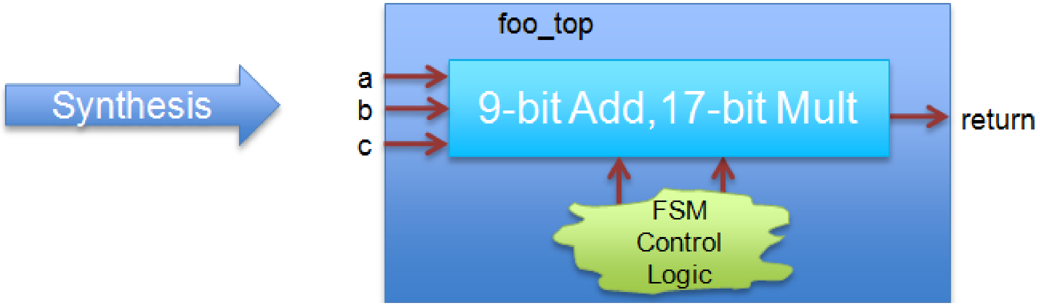
- **Code using native C int type**

```
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum*c;
    return mult;
}
```



- **However, if the inputs will only have a max range of 8-bit**
  – Arbitrary precision data-types should be used

```
int17 foo_top(int8 a, int8 b, int8 c)
{
    int9 sum;
    int17 mult
    sum=a+b;
    mult=sum*c;
    return mult;
}
```



  – It will result in smaller & faster hardware with the full required precision
  – With arbitrary precision types on function interfaces, Vivado HLS can propagate the correct bit-widths throughout the design

HLS & C Types

- **There are 4 basic types you can use for HLS**
  - Standard C/C++ Types
  - Vivado HLS enhancements to C: apint
  - Vivado HLS enhancements to C++: ap_int, ap_fixed
  - SystemC types

Type of C	C(C99) / C++	Vivado HLS ap_cint (bit-accurate with C)	Vivado HLS ap_int (bit-accurate with C++)	OSCI SystemC (IEEE 1666-2005 :bit-accurate)
Description		Used with standard C	Used with standard C++	IEEE standard
Requires		#include "ap_cint.h"	#include "ap_int.h" #include "ap_fixed.h" #include "hls_stream.h"	#include "systemc.h"
Pre-Synthesis Validation	gcc/g++		g++	g++
			Vivado HLS GUI	Vivado HLS GUI
Fixed Point	NA	NA	ap_fixed	#define SC_INCLUDE_FX sc_fixed
Signal Modeling	Variables	Variables	Variables Streams	Signals, Channels, TLM (1.0)

Arbitrary Precision : C++ ap_fixed types

- **Support for fixed point datatypes in C++**
  - Include the path to the ap_fixed.h header file
  - Both signed (ap_fixed) and unsigned types (ap_ufixed)

```
#include ap_fixed.h                    $VIVADO_HLS_HOME/include/ap_fixed.h

void foo_top (…) {

 ap_fixed<9, 5, AP_RND_CONV, AP_SAT>  var1;          //  9-bit,
                                                     //  5 integer  bits, 4 decimal places

 ap_ufixed<10, 7, AP_RND_CONV, AP_SAT>  var2;        // 10-bit unsigned
                                                     //   7 integer bits, 3 decimal places
```

- **Advantages of Fixed Point types**
  - The result of variables with different sizes is automatically taken care of
  - The binary point is automatically aligned
    - Quantization: Underflow is automatically handled
    - Overflow: Saturation is automatically handled

Alternatively, make the result variable large enough such that overflow or underflow does not occur

Definition of ap_fixed type

- **Fixed point types are specified by**
  - Total bit width (W)
  - The number of integer bits (I)
  - The quantization/rounding mode (Q)
  - The overflow/saturation mode (O)

ap_[u]fixed<W, I , Q, O , N>

I-1	…	1	0	-1	…	-B

Binary point : W = I + B

	Description	
W	Word length in bits	
I	The number of bits used to represent the integer value (the number of bits above the decimal point)	
Q	Quantization mode (modes detailed below) dictates the behavior when greater precision is generated than can be defined by the LSBs.	
	AP_Fixed Mode	**Description**
	AP_RND	Rounding to plus infinity
	AP_RND_ZERO	Rounding to zero
	AP_RND_MIN_INF	Rounding to minus infinity
	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity
	AP_TRN_ZERO	Truncation to zero (default)
O	Overflow mode (modes detailed below) dictates the behavior when more bits are required than the word contains.	
	AP_Fixed Mode	**Description**
	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	The number of saturation bits in wrap modes.	

Quantization Modes

- **Quantization mode**
  - Determines the behavior when an operation generates more precision in the LSBs than is available
- **Quantization Modes (rounding):**
  - AP_RND, AP_RND_MIN_IF, AP_RND_IF
  - AP_RND_ZERO, AP_RND_CONV
- **Quantization Modes (truncation):**
  - AP_TRN, AP_TRN_ZERO

Quantization Modes: Truncation

- **AP_TRN: truncate**
  - Remove redundant bits. Always rounds to minus infinity
  - This is the default.
    - 01.01(1.25) ➔ 01.0 (1)

- **AP_TRN_ZERO: truncate to zero**
  - For positive numbers, the same as AP_TRN
    - For positive numbers: 01.01(1.25) ➔ 01.0(1)
  - For negative numbers, round to zero
    - For negative numbers: 10.11 (-1.25) ➔ 11.0(-1)

Overflow Modes

- **Overflow mode**
  - Determines the behavior when an operation generates more bits than can be satisfied by the MSB
- **Overflow Modes (saturation)**
  - AP_SAT, AP_SAT_ZERO, AP_SAT_SYM
- **Overflow Modes (wrap)**
  - AP_WRAP, AP_WRAP_SM
  - The number of saturation bits, N, is considered when wrapping

Overflow Mode: Saturation

- **AP_SAT: saturation**
  - This overflow mode will convert the specified value to MAX for an overflow or MIN for an underflow condition
  - MAX and MIN are determined from the number of bits available
- **AP_SAT_ZERO: saturates to zero**
  - Will set the result to zero, if the result is out of range
- **AP_SAT_SYM: symmetrical saturation**
  - In 2’s complement notation one more negative value than positive value can be represented
  - If it is desirable to have the absolute values of MIN and MAX symmetrical around zero, AP_SAT_SYM can be used
  - Positive overflow will generate MAX and negative overflow will generate -MAX
    - 0110(6) => 011(3)
    - 1011(-5) => 101(-3)

AP_FIXED operators & conversions

Fully Supported for all Arithmetic operator

<table>
<tr><th colspan="2">Operations</th></tr>
<tr><td>Arithmetic</td><td>+ - * / % ++ --</td></tr>
<tr><td>Logical</td><td>~ !</td></tr>
<tr><td>Bitwise</td><td>& | ^</td></tr>
<tr><td>Relational</td><td>> < <= >= == !=</td></tr>
<tr><td>Assignment</td><td>*= /= %= += -=
<<= >>= &= ^= |=</td></tr>
</table>

Methods for type conversion

Methods		Example
To integer	Convert to a integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.double();
To ap_int	Convert to an ap_int	res = var.to_ap_int();

AP_FIXED methods

Methods for bit manipulation

Methods		Example
Length	Returns the length of the variable.	res=var.length;
Concatenation	Concatenation low to high	res=var_hi.concat(var_lo); Or res= (var_hi,var_lo)
Range or Bit-select	Return a bit-range from high to low or a specific bit.	res=var.range(high bit,low bit); Or res=var[bit-number]

Fixed Point Math Functions

- **The hls_math.h library**
  - Now includes fixed-point functions for sin, cos and sqrt

Function	*Type*	Accuracy (ULP)	Implementation Style
cos	ap_fixed<32,I>	16	Synthesized
sin	ap_fixed<32,I>	16	Synthesized
sqrt	ap_fixed<W,I> ap_ufixed<W,I>	1	Synthesized

  - The sin and cos functions are all 32-bit ap_fixed<32,Int_Bit>
    - Where Int_Bit specifies the number of integer bits
  - The sqrt function is any width but must have a decimal point
    - Cannot be all intergers or all bits
  - The accuracy above is quoted with respect to the equivalent floating point version

Floating Point Support

- **Synthesis for floating point**
  - Data types (IEEE-754 standard compliant)
    - Single-precision
      - 32 bit: 24-bit fraction, 8-bit exponent
    - Double-precision
      - 64 bit: 53-bit fraction, 11-bit exponent
- **Support for Operators**
  - Vivado HLS supports the Floating Point (FP) cores for each Xilinx technology
    - If Xilinx has a FP core, Vivado HLS supports it
    - It will automatically be synthesized
  - If there is no such FP core in the Xilinx technology, it will not be in the library
    - The design will be still synthesized

Floating Point Cores

Core	7 Series	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
FRSqrt	X	X	X	-	-	-
FRSqrt_nodsp	X	X	X	-	-	-
FRSqrt_fulldsp	X	X	X	-	-	-
FRecip	X	X	X	-	-	-
FRecip_nodsp	X	X	X	-	-	-
FRecip_fulldsp	X	X	X	-	-	-
FSqrt	X	X	X	X	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X
DRSqrt	X	X	X	X	X	X
DRecip	X	X	X	-	-	-
DSqrt	X	X	X	-	-	-

Support for Math Functions

More Details are available in the Coding Style Guide chapter in the User Guide

- **Vivado HLS provides support for many math functions**
  - Even if no floating-point core exists
  - These functions are implemented in a bit-approximate manner
  - The results may differ within a few Units of Least Precision (ULP) to the C/C++ standards
- **Use math.h (C) or cmath.h (C++)**
  - The functions will be synthesized automatically
  - The C simulation results may differ from the RTL simulation results
  - Use a test bench which checks for ranges: not == or !=
- **Replace math.h or cmath.h with Vivado HLS header file "hls_math.h" Or keep math/cmath and "add_files hls_lib.c"**
  - The C simulation will match the RTL simulation
  - The C simulation may differ from the C simulation using math/cmath (or math/cmath without hls_lib.c)

Supported Math Functions

- **Floating C point functions ***f**
  - There is no double-precision implementation
  - C++ functions will overload as per the C++ standard
    - Can be used with double or single precision
- **More specific details are in the User Guide**
  - Refer to the Coding Style Guide chapter: C Libraries

Function	Float	Double	Accuracy (ULP)	LogicCore
ceilf	Supported	Not Applicable	Exact	Not Supported
copysignf	Supported	Not Applicable	Exact	Not Supported
fabsf	Supported	Not Applicable	Exact	Not Supported
floorf	Supported	Not Applicable	Exact	Not Supported
logf	Supported	Not Applicable	1 to 5	Not Supported
cosf	Supported	Not Applicable	1 to 100	Not Supported
sinf	Supported	Not Applicable	1 to 100	Not Supported
abs	Supported	Supported	Exact	Not Supported
ceil	Supported	Supported	Exact	Not Supported
copysign	Supported	Supported	Exact	Not Supported
cos	Supported	Supported	2 for float. 5 for double	Not Supported
fabs	Supported	Supported	Exact	Not Supported
floor	Supported	Supported	Exact	Not Supported
fpclassify	Supported	Supported	Exact	Not Supported
isfinite	Supported	Supported	Exact	Not Supported
isinf	Supported	Supported	Exact	Not Supported
isnan	Supported	Supported	Exact	Not Supported
isnormal	Supported	Supported	Exact	Not Supported
log	Supported	Supported	1 for float, 16 for double	Not Supported
log10	Supported	Supported	1 for float, 16 for double	Not Supported
recip	Supported	Supported	Exact	Supported
round	Supported	Supported	Exact	Not Supported
rsqrt	Supported	Supported	Exact	Supported
signbit	Supported	Supported	Exact	Not Supported
sin	Supported	Supported	2 for float, 5 for double	Not Supported
sqrt	Supported	Supported	Exact	Supported
trunc	Supported	Supported	Exact	Not Supported

For more information on floating point refer to Application Note **Floating Point Design with Vivado HLS**

Example on using Floating Point Types

- **The following highlights some typical use scenarios**
  - Example values

```
double          foo_d  = 3.1459;
float           foo_f  = 3.1459;
ap_fixed<14,4>  foo_fx = -1.4142;
int             foo_i  = 42;
```

Using ap_fixed requires:
- C++
- $Vivado HLS_HOME/include/ap_fixed.h

- **When using sqrt() function**
  - It is from math.h which is a C function, not C++

```
extern "C" float sqrtf(float);
```

Required if it's a C++ function

- **Understand that sqrt() is 64-bit and sqrtf() is 32-bit**

```
double   var_d = sqrt(foo_d);     // 64-bit sqrt core
float    var_f = sqrtf(foo_f);    // This will lead to a single precision sqrt core

         var_f = sqrt(foo_f);     // Still 64-bit, with format conversion cores (single to double and back)
```

- **Type conversions can be used**

```
ap_fixed<14,4>   var_fx = sqrtf(foo_fx);   // fixed-point to single precision conversion
                                           // Fixed → 32-bit sqrt core → float to fixed conversion
int              var_i = sqrtf(foo_i);     // int to float conversion
                                           // Int → 32-bit sqrt → float to int
```

Using sqrt instead of sqrtf would imply a single to double conversion and back

Pipelines and Dependencies – Wolf Chapter 6

- Before we start on QR, some background on what Vitis HLS is trying to do:
  - ⇒ See: **wolf_ch6c.pdf**
  - ⇒ In Books_Readings: Wolf_Book

- Data dependencies describe relationships between operations:
  - ⇒ x <= a + b; value of x depends on a, b
- High-level synthesis must preserve data dependencies.

Data Flow Graph

- Data flow graph (DFG) models data dependencies.
- Does not require that operations be performed in a particular order.
- Models operations in a basic block of a functional model—no conditionals.
- Requires single-assignment form.

Data Flow Graph Construction

original code:

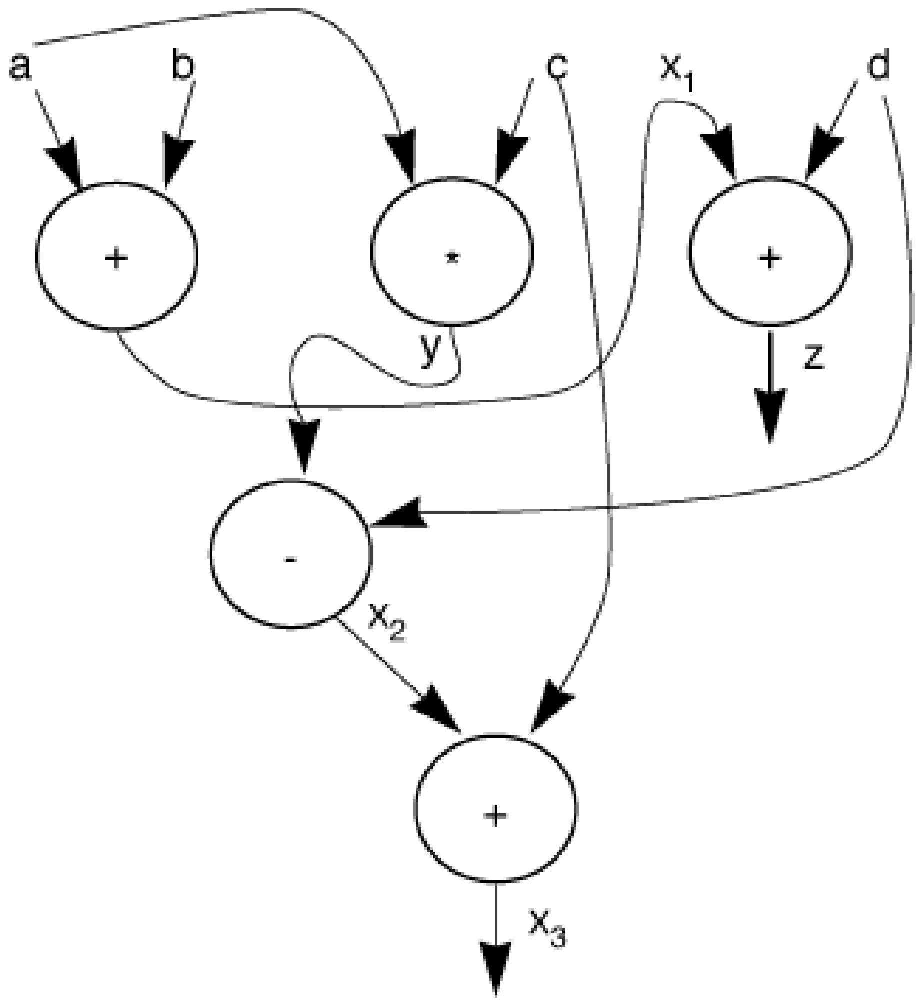
```
x <= a + b;
y <= a * c;
z <= x + d;
x <= y - d;
x <= x + c;
```

single-assignment form:

```
x1 <= a + b;
y <= a * c;
z <= x1 + d;
x2 <= y - d;
x3 <= x2 + c;
```

Data flow graph construction, cont'd

Data flow forms directed acyclic graph (DAG):

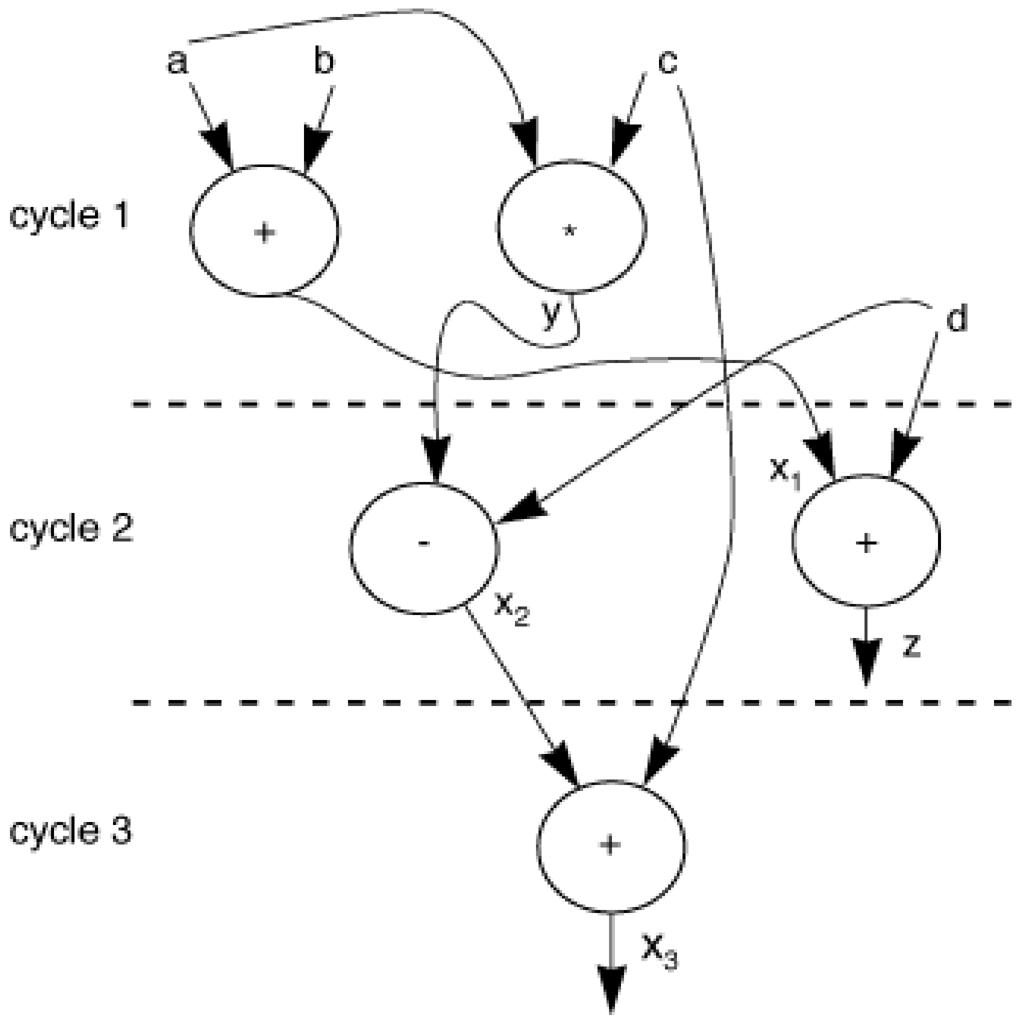


Goals of Scheduling and Allocation

- Preserve behavior—at end of execution, should have received all outputs, be in proper state (ignoring exact times of events).
- Utilize hardware efficiently.
- Obtain acceptable performance.

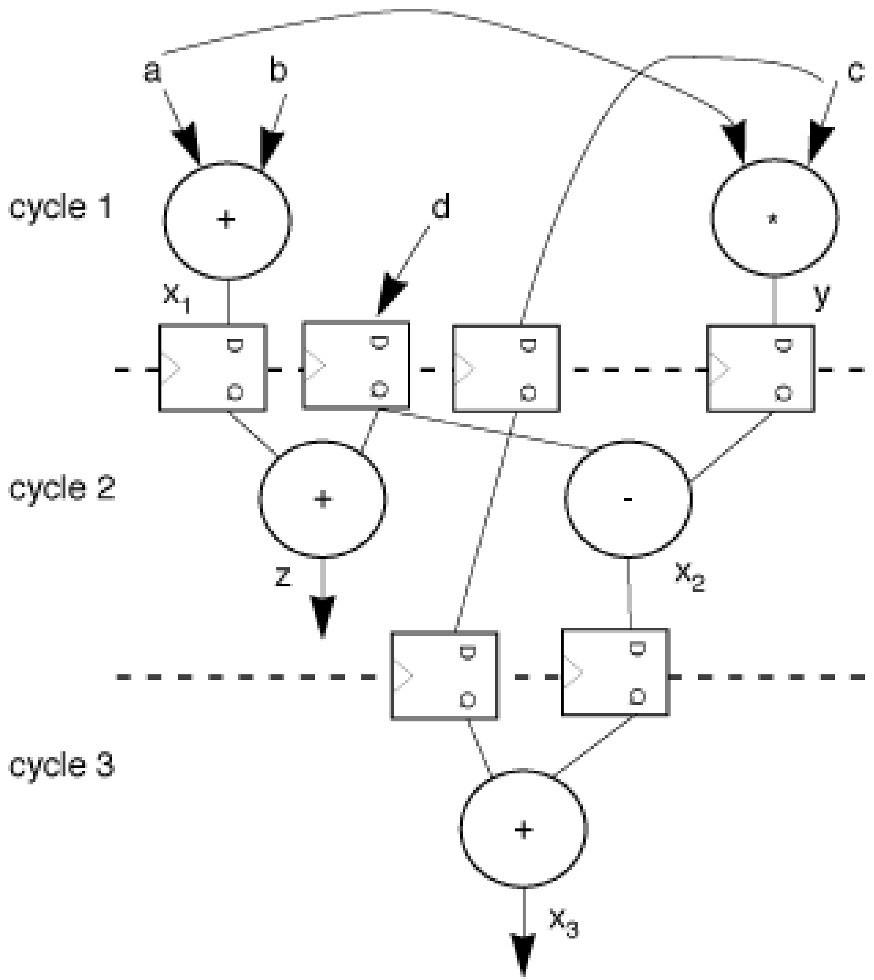
Data flow to data path-controller

One feasible schedule for last DFG:



Binding values to registers

Registers placed on clock cycle boundaries



Initial Ten Register lifetimes

Layer 1	R1 A	R2 B	R3 C	R4 D
Layer 2	X1	D	C	Y
Layer 3			C’	X2

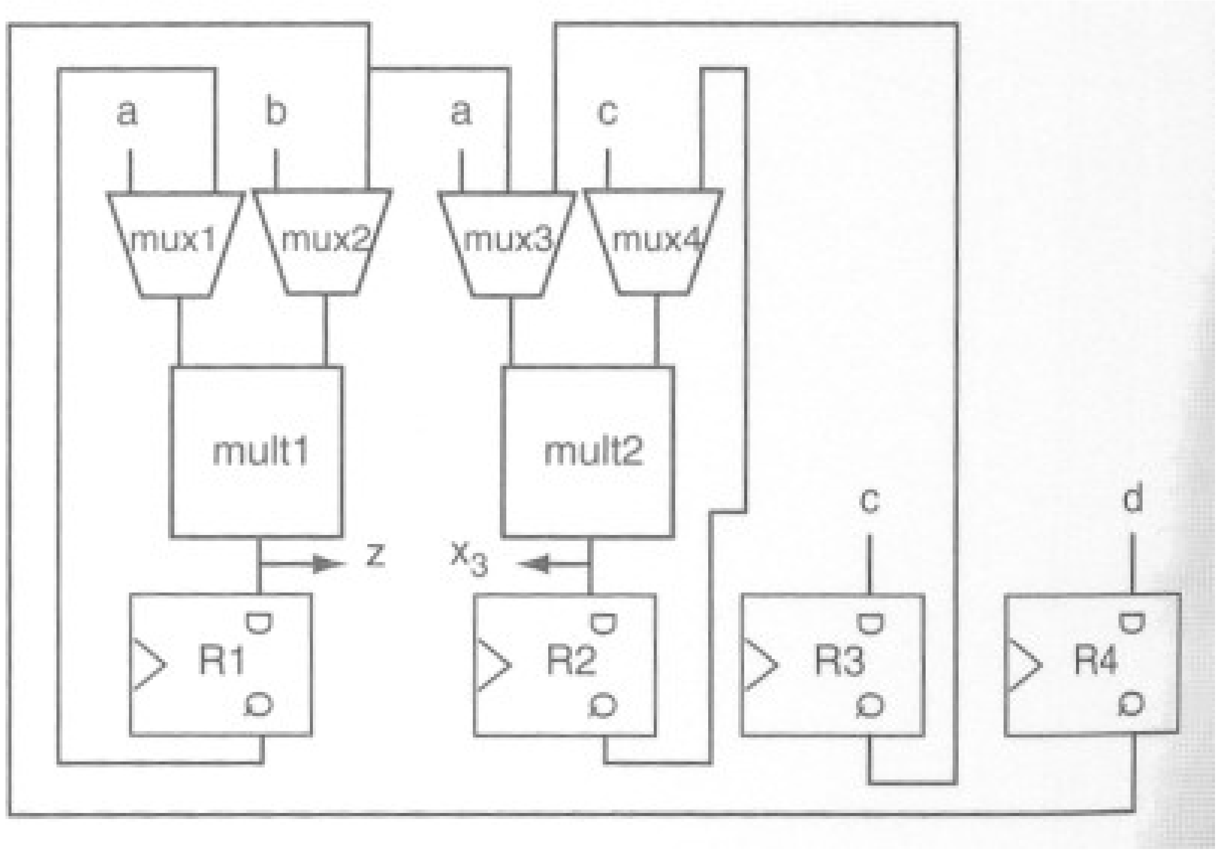
Allocation Could “fold” with multiplexers

- Instead of Fully Pipelined with 10 registers, if resources are an issue, one could fold and reuse.
- Same unit used for different values at different times.
  - ⇒ Function units.
  - ⇒ Registers.
- Multiplexer controls which value has access to the unit.

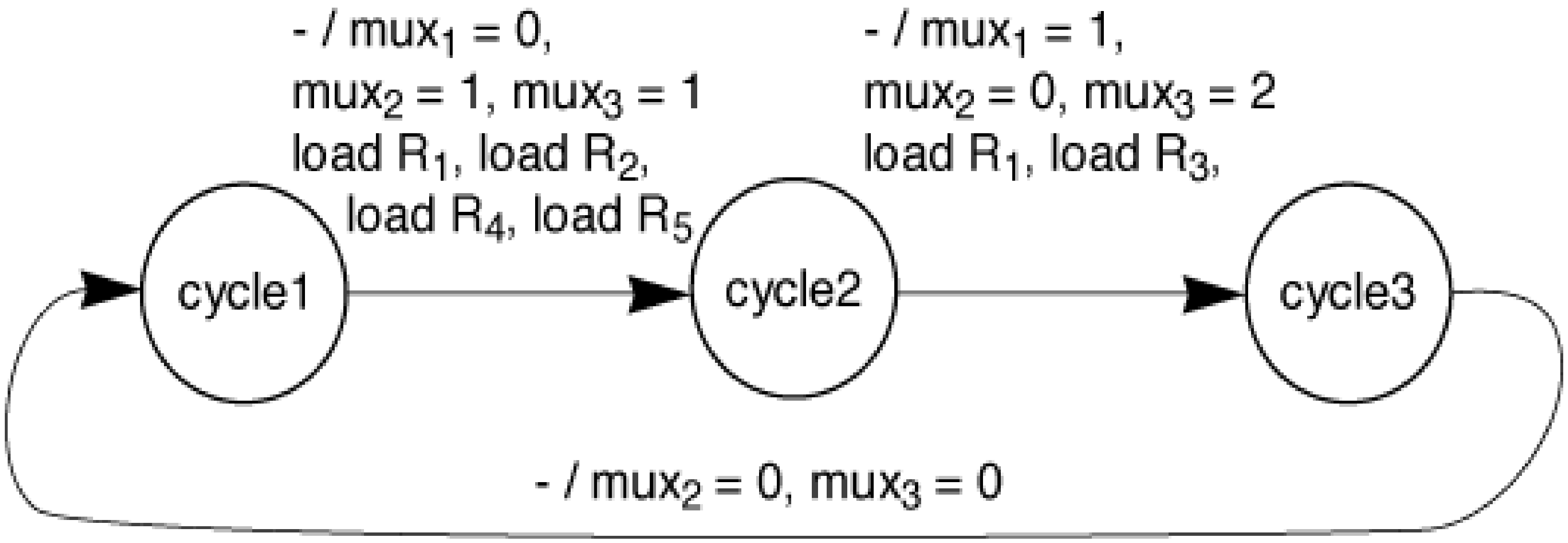
Choosing function units – Folded design

Muxes allow function units to be shared for several operations (Mult & Add)

Simplified as p. 382 to 4 Registers



Folded Design Needs a FSM sequencer



Sequencer requires three states, even with no conditionals

Choices during high-level synthesis

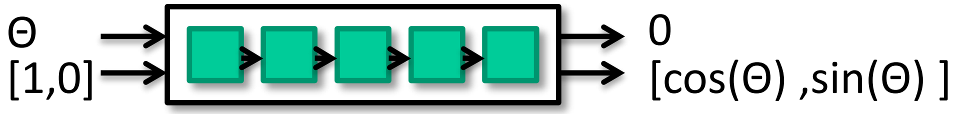
- Scheduling determines number of clock cycles required; binding determines area, cycle time.
- Area tradeoffs must consider shared function units vs. multiplexers, control.
- Delay tradeoffs must consider cycle time vs. number of cycles.

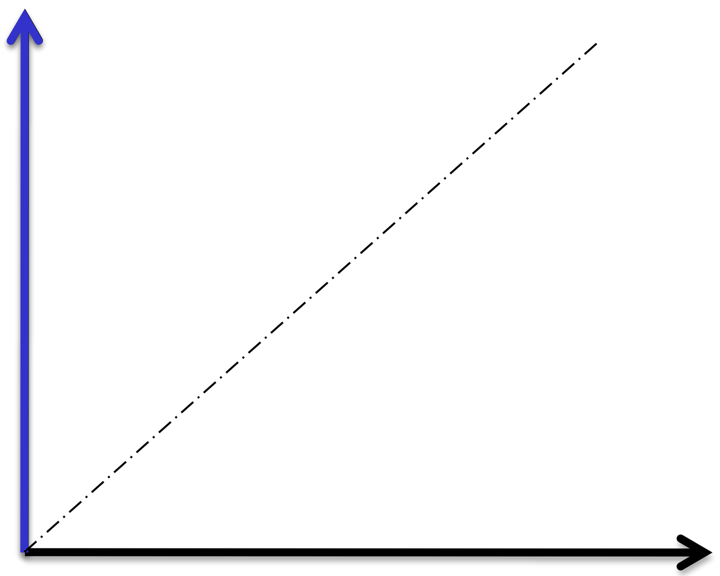
Review of CORDIC algorithm

- Cheap iterative algorithm – iterative successive unitary micro-rotations
- Blocks useful to us:
  - $\Rightarrow$ COS/SIN
  - $\Rightarrow$ ATAN

Review of CORDIC algorithm

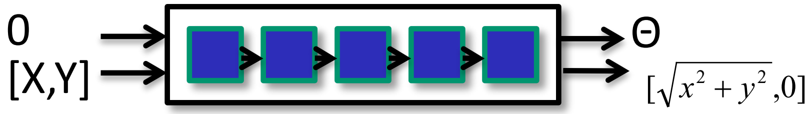
- COS/SIN Block
  - ⇒ Input: Θ, Initial vector [1,0]
  - ⇒ Output: sin(Θ), cos(Θ)
  - ⇒ Apply successively smaller unitary micro-rotations

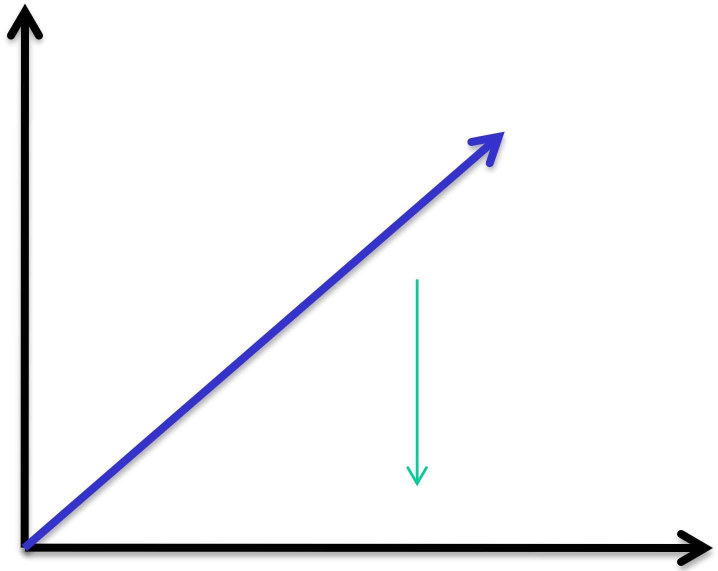




Review of CORDIC algorithm

- ATAN Block
  - ⇒ Input: Θ, Initial vector [X,Y], initial angle 0
  - ⇒ Output: Θ, [R,0]
  - ⇒ Apply successively smaller unitary micro-rotations





QR Decomposition

- Decompose matrix into Q and R component

$$A = QR$$

$$A = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

- Number of methods: Gram-Schmidt, Householder, Givens rotation

  $\Rightarrow$ Successive unitary transformations

$$R = Q_6{}^T \bullet \ldots \bullet Q_2{}^T \bullet Q_1{}^T \bullet A$$

$$R = Q^T A$$

QR Decomposition

- Decompose matrix into Q and R component

$$A = QR$$

$$A = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

- Number of methods: Gram-Schmidt, Householder, ***Givens rotation***

$\Rightarrow$ Successive unitary transformations to induce zeros

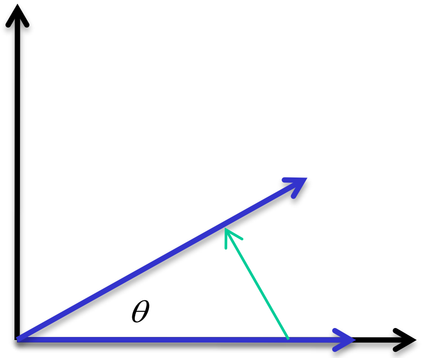
$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$

$$R = Q^T A$$

Givens Rotation

- Rotation matrix

$$T = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

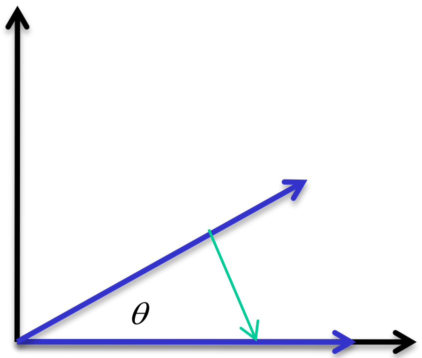


- Givens Rotation Matrix

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

$$r = \sqrt{a^2 + b^2}$$

$$\theta = \tan^{-1}\left(b/a\right)$$



Order of Rotation, n = 8

x							
7	x						
6	9	x					
5	8	11	x				
4	7	10	13	x			
3	6	9	12	15	x		
2	5	8	11	14	17	x	
1	4	7	10	13	16	19	x

Rotation pair

$$
\begin{array}{cc}
 & \begin{array}{c}\text{col. } i\\ \downarrow\end{array} \\
P_{i+1,j} = & \begin{bmatrix}
1 & & & \vdots & & & & \\
 & \ddots & & \vdots & & & & \\
 & & & c_i & s_i & \cdots & & \\
 & & & -s_i & c_i & & & \\
 & & & & & \ddots & & \\
 & & & & & & & 1
\end{bmatrix} \leftarrow \text{row } i.
\end{array}
$$

QR Decomposition via Givens Rotations

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$

$$R = Q^T A$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

QR Decomposition via Givens Rotations

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$
$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

QR Decomposition via Givens Rotations

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$
$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

QR Decomposition via Givens Rotations

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$

$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & 0 & r_{43} & r_{44} \end{bmatrix}$$

Parallelizing Operation

- Successive unitary transformations to induce zeros

$$R = Q_6^{\,T} \bullet \ldots \bullet Q_2^{\,T} \bullet Q_1^{\,T} \bullet A$$

$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

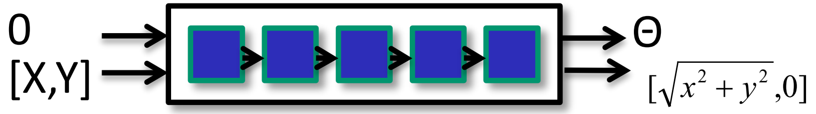
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & 0 & r_{43} & r_{44} \end{bmatrix}$$

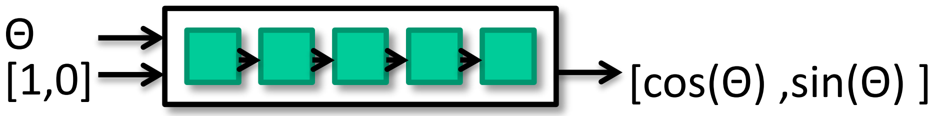
Givens Rotation and CORDIC

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

- Vectoring Mode



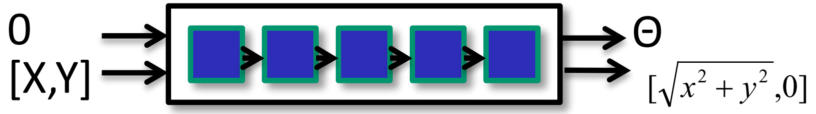
- Rotation Mode



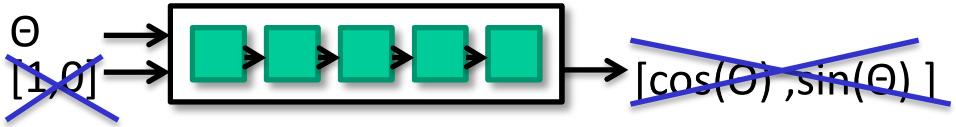
Givens Rotation and CORDIC

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

- Vectoring Mode



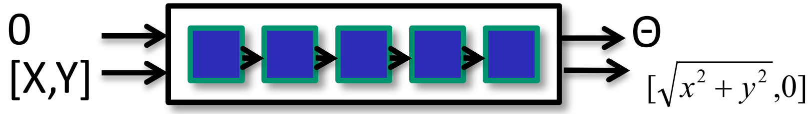
- Rotation Mode



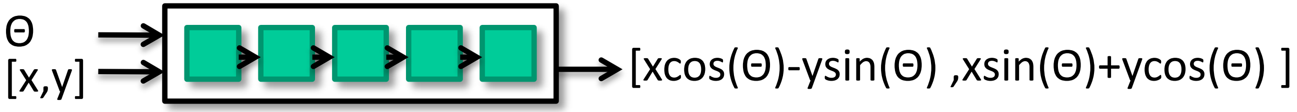
Givens Rotation and CORDIC

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

- Vectoring Mode



- Rotation Mode



Parallelizing Operation

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$

$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ 0 & r_{32} & r_{33} & r_{34} \\ 0 & 0 & r_{43} & r_{44} \end{bmatrix}$$

Parallelizing Operation

- Overlap Different Matrix Multiplications

$$A = QR$$

$$A = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

Parallelizing Operation

- Overlap Different Matrix Multiplications
- Problem: Dependencies

$$A = QR$$

$$A = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

Parallelizing Operation

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$
$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = ???$$

Parallelizing Operation

- Successive unitary transformations to induce zeros

$$R = Q_6^T \bullet \ldots \bullet Q_2^T \bullet Q_1^T \bullet A$$
$$R = Q^T A$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ 0 & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{41} & 0 & r_{43} & r_{44} \end{bmatrix}$$

Can not apply this transformation

Next Lecture

- More on QR Decomposition Scheduling
- Project 4 CORDIC discussion