

Lecture 1: Introduction

Sitao Huang

sitaoh@uci.edu

January 4, 2022



Instructor

Sitao Huang

Assistant Professor, EECS

Email: sitaoh@uci.edu

Website: sitaohuang.com

- Research Interests:
 - *Hardware accelerators*
 - *Programming languages and compilers for accelerators*
 - *High-level synthesis (HLS)*
 - *Heterogeneous computing*
- *Ph.D., University of Illinois at Urbana-Champaign (2021)*
- *M.S., University of Illinois at Urbana-Champaign (2017)*
- *B.S., Tsinghua University (2014)*



Recruiting students!

Languages and Compilers for Hardware Accelerators

- Topics
 - Hardware accelerators (FPGAs, GPUs, ASICs, etc.)
 - Types, design methodology, performance metrics, etc.
 - Programming languages for accelerators
 - Abstraction levels, language features, programming paradigms, etc.
 - Compilation flows for accelerators
 - Compiler construction, optimizations, hardware-specific considerations, etc.
 - Recent works in the field
- Goals
 - Get a better understanding on the hardware accelerator design flows
 - Know about various ways of designing languages and compilers for accelerators
 - Compiler optimization techniques for hardware accelerators
 - Get to know state-of-the-art research works in this field

Announcements, Time, and Location

- **Announcements:** Check [Canvas](#) and emails for the latest announcements
- **Lecture Time:**
 - Tuesdays and Thursdays 8:00 – 9:20 am
- **Lecture Location:**
 - First two weeks (1/3 – 1/14) or until further notice: *online*, [Zoom link](#)
 - Later weeks if in-person: [SSTR 101](#)
- **Office Hours:**
 - Tuesdays 9:30 – 10:30 am or by appointment (sitaoh@uci.edu)
 - Location: [Lecture Zoom link](#), if lectures are online
Engineering Hall 3215, if lectures are in-person

Grading Policy

- Homework – 30%
 - Four assignments, each 7.5%
- Midterm – 30%
 - February 10 (Thursday, Week 6), in class
- Course Project – 40%
 - Proposal (due Jan. 31), 10%
 - Final Presentation (Week 10), 15%
 - Final Report (Week 10), 15%

NOTE: Please check Canvas for the latest announcements

Tentative Schedule

- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compiler Optimizations for Accelerators
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

Languages and Compilers for Hardware Accelerators

- *What?*
- *Why?*
- *How?*

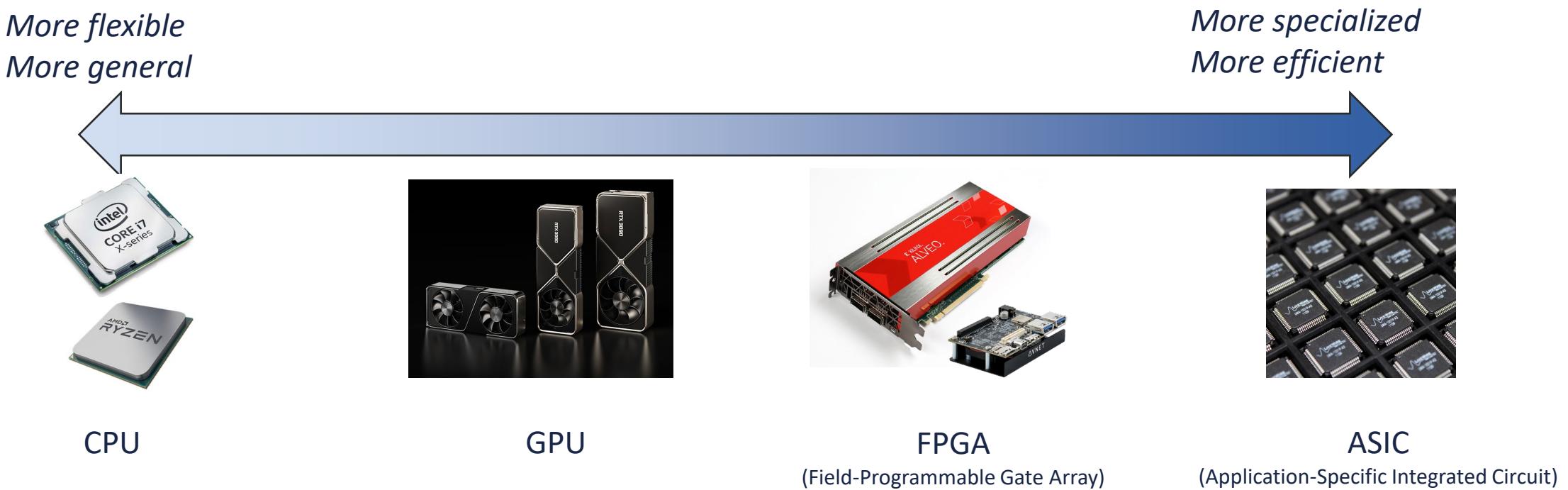
Languages and Compilers for Hardware Accelerators

- *What?*
- *Why?*
- *How?*

Languages and Compilers for Hardware Accelerators

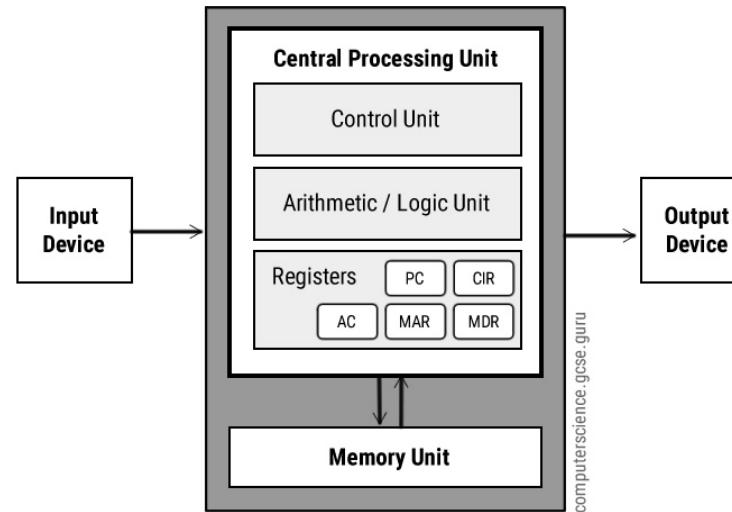
Languages and Compilers for *Hardware Accelerators*

- **Hardware accelerator:** “computer hardware designed to perform specific functions more efficiently compared to software running on a CPU” (Wikipedia)
- Tradeoff between flexibility and efficiency
- Invest time and money for better performance and efficiency



General Purpose Processor

- Programmable processors used in a variety of applications
- Not designed for any specialized purposes
- Central Processing Unit (CPU)
- Architecture (Von Neuman)
 - Arithmetic Logic Unit (ALU)
 - Control Unit
 - Registers
 - Memory management unit (MMU)
 - Cache
- Implemented on Integrated Circuit (IC)
 - one or more CPU cores in a single IC chip
- An IC that contains a CPU may also contain memory, peripheral interfaces
 - Microcontrollers and System-on-Chip (SoC)



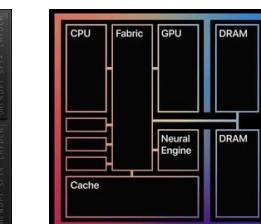
Intel i9-12900K CPU



Qualcomm Snapdragon SoC



Microcontroller Unit (MCU)
on an Arduino board

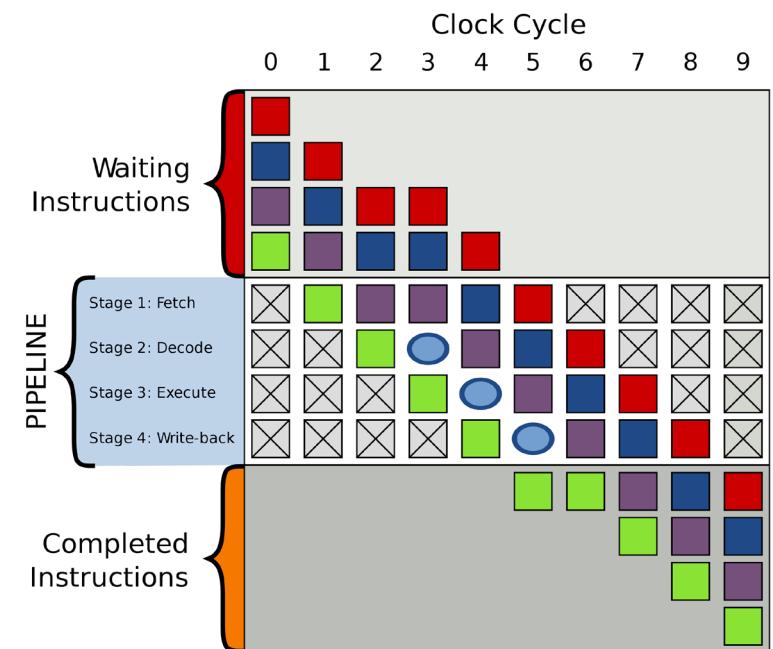


Apple M1 SoC

General Purpose Processor

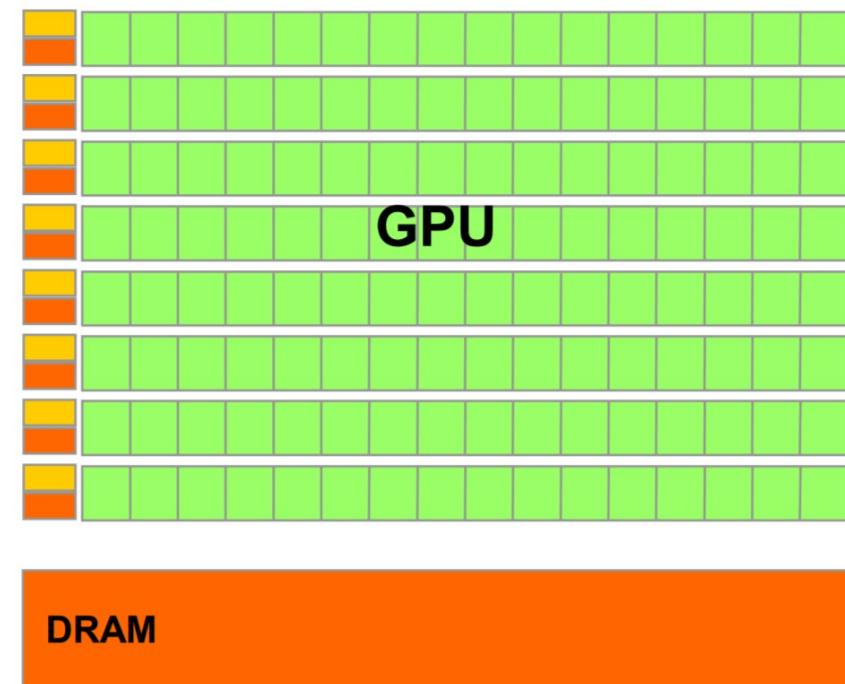
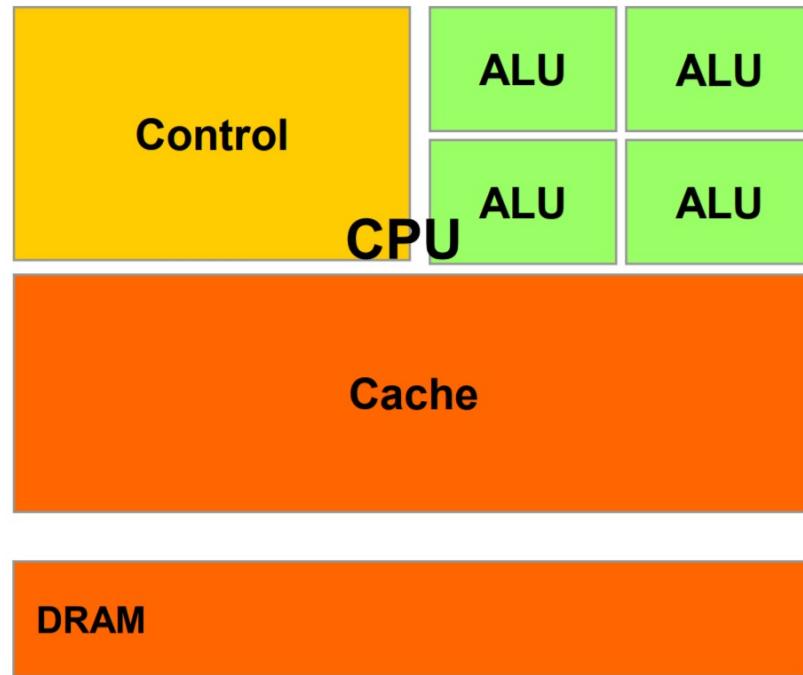
- Instruction Pipelining
 - A technique for implementing instruction-level parallelism within a processor
 - Pipeline stages (five-stage pipeline case)
 - Instruction Fetch (IF)
 - Instruction Decode (ID)
 - Execute (EX)
 - Memory access (MEM)
 - Register write back (WB)
- Hazards
 - *Data hazard*: values produced from one instruction are not available when needed by a subsequent instruction
 - *Control hazard*: a branch in the control flow makes ambiguous what is the next instruction to fetch
- Pipeline Stall
 - Delay in execution of an instruction in order to resolve a hazard

Instr. No.	Clock cycle	Basic five-stage pipeline						
		1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB			
2		IF	ID	EX	MEM	WB		
3		IF	ID	EX	MEM	WB		
4			IF	ID	EX	MEM		
5				IF	ID	EX		



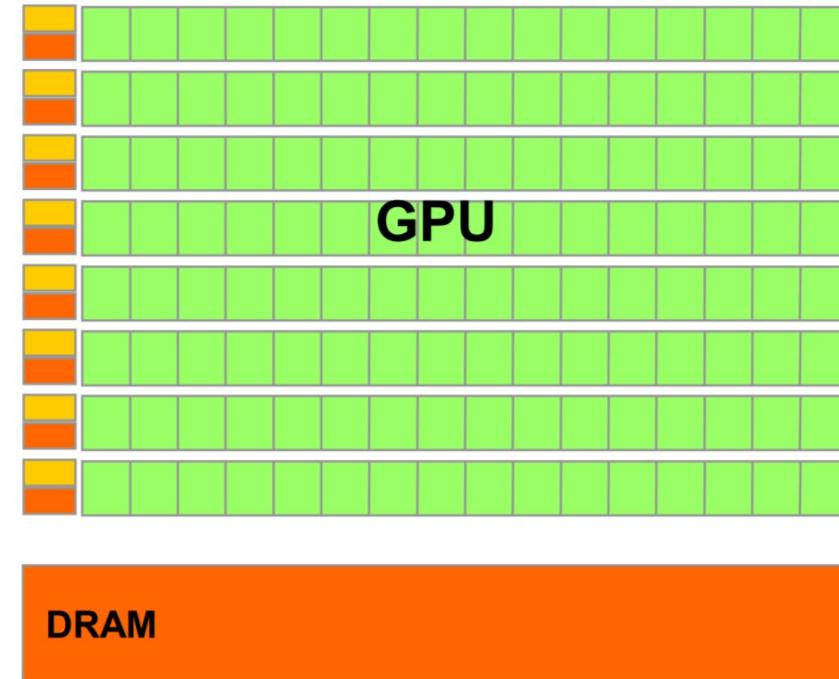
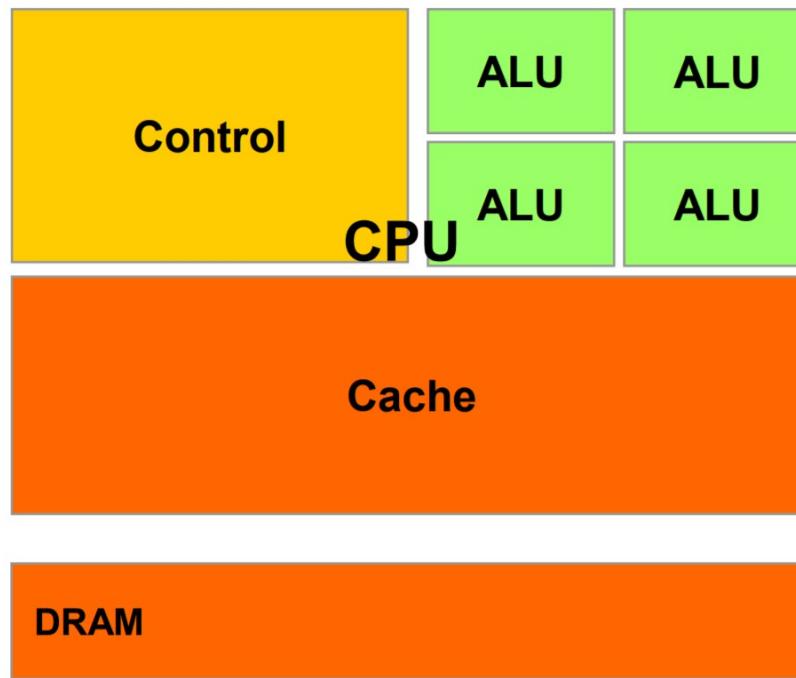
Graphics Processing Unit (GPU)

- CPUs and GPUs have fundamentally different design philosophies
 - CPUs: *low*-latency, *low*-throughput
 - high clock freq., large caches, sophisticated control, powerful ALUs
 - GPUs: *high*-latency, *high*-throughput
 - moderate clock freq., small caches, simple control, (many) energy efficient ALUs
 - Require massive number of threads to tolerate latencies* → **More specialized in specific applications**



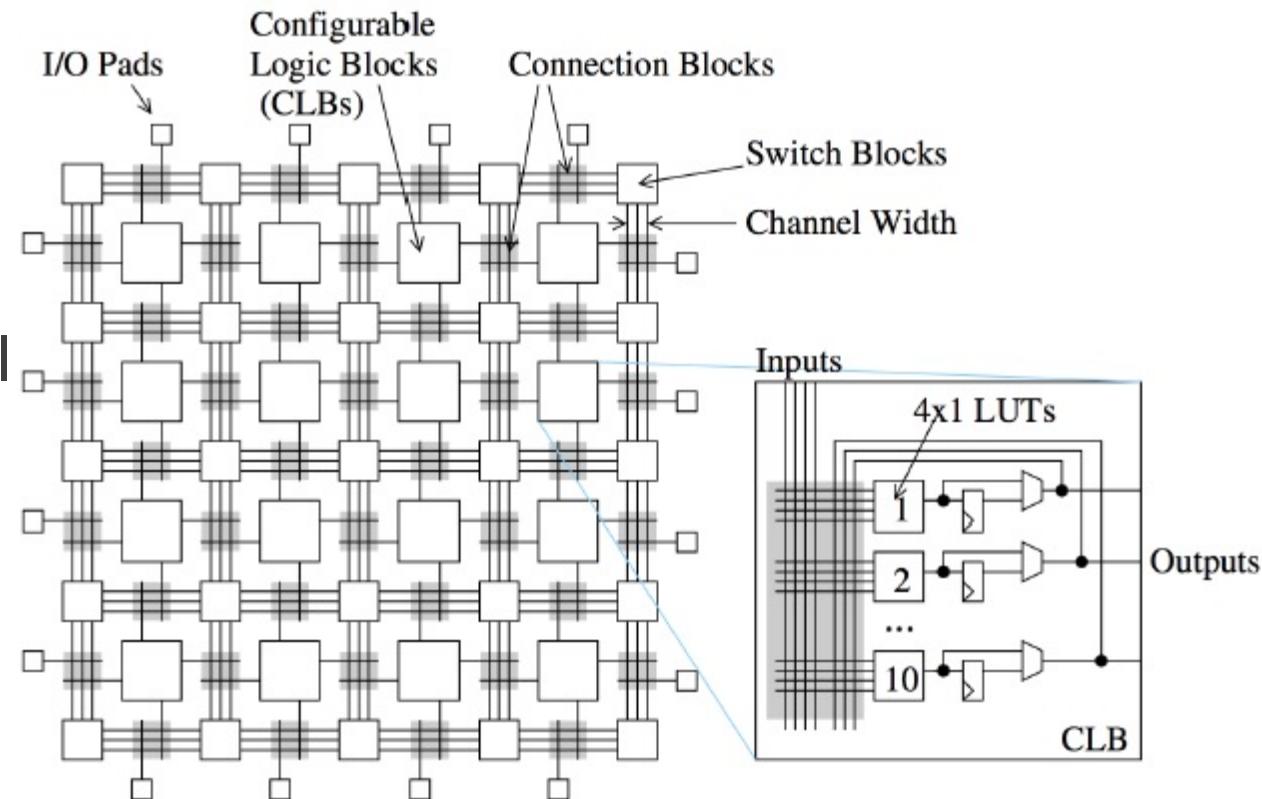
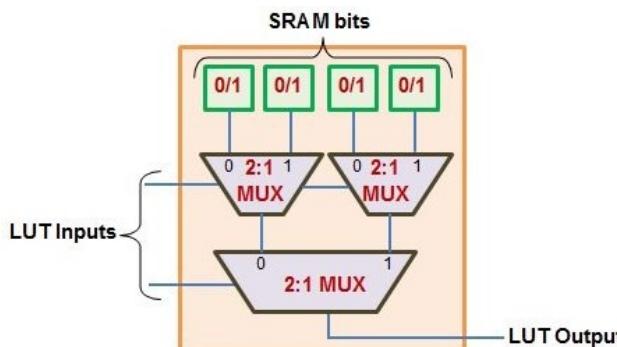
Graphics Processing Unit (GPU)

- CPUs and GPUs have fundamentally different design philosophies
 - CPUs: Good for sequential parts where latency matters
 - CPUs can be > 10x faster than GPUs for sequential code
 - GPUs: Good for parallel parts where throughput wins
 - GPUs can be > 10x faster than CPUs for parallel code



Field-Programmable Gate Array (FPGA)

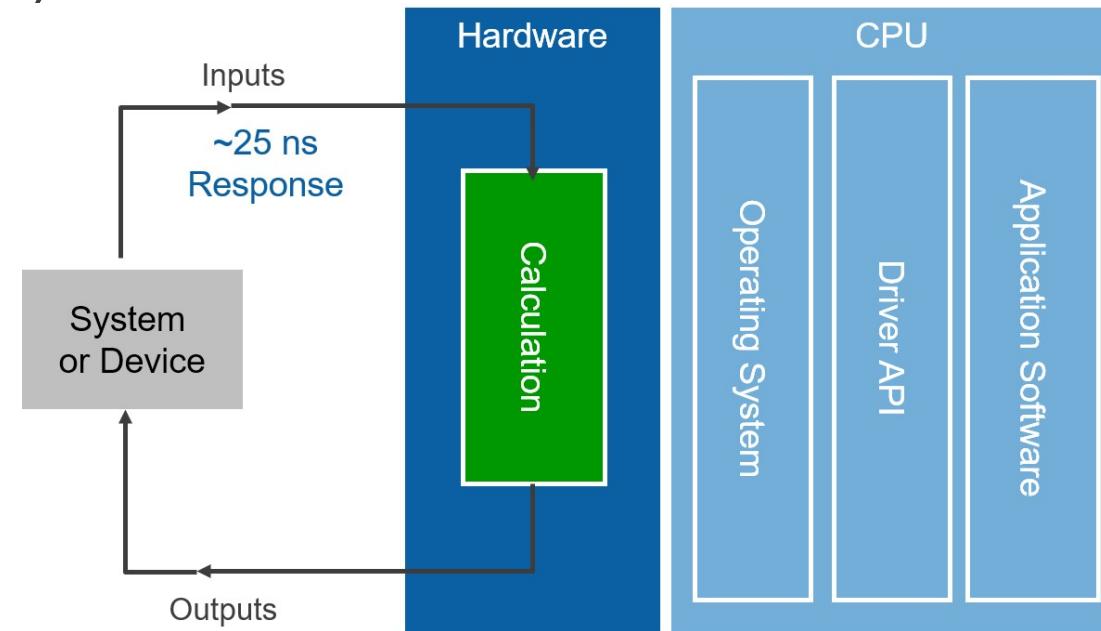
- High-density programmable gate array
- Fully programmable through bit-stream configuration file
 - Programmable Logic
 - Programmable I/O
 - Programmable Interconnects (routing)
- Look-up table (LUT) based combinational logic
 - Can be implemented with SRAM (volatile)



Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

Field-Programmable Gate Array (FPGA)

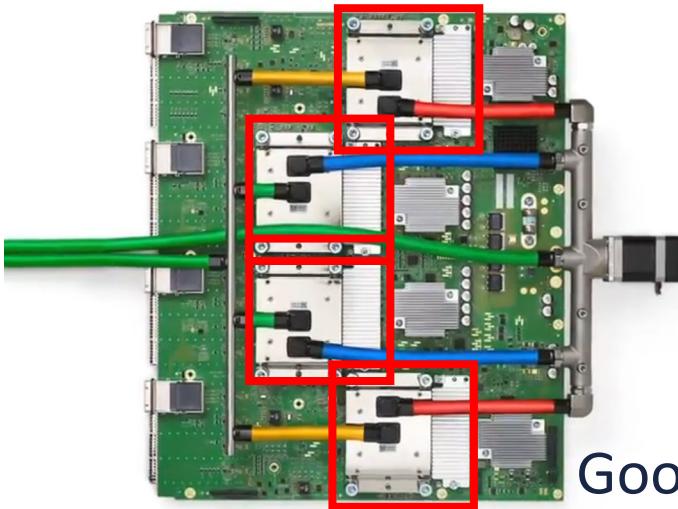
- Massive fine-grained parallelism
- Clock cycle accurate control & compute
- Very low latency, very short response time
(benefits from specialization and customization)
 - Applications: real-time video processing, signal processing, high-frequency trading, etc.
- Lower clock frequency (typically < 1GHz) compared to CPU/GPU/ASIC



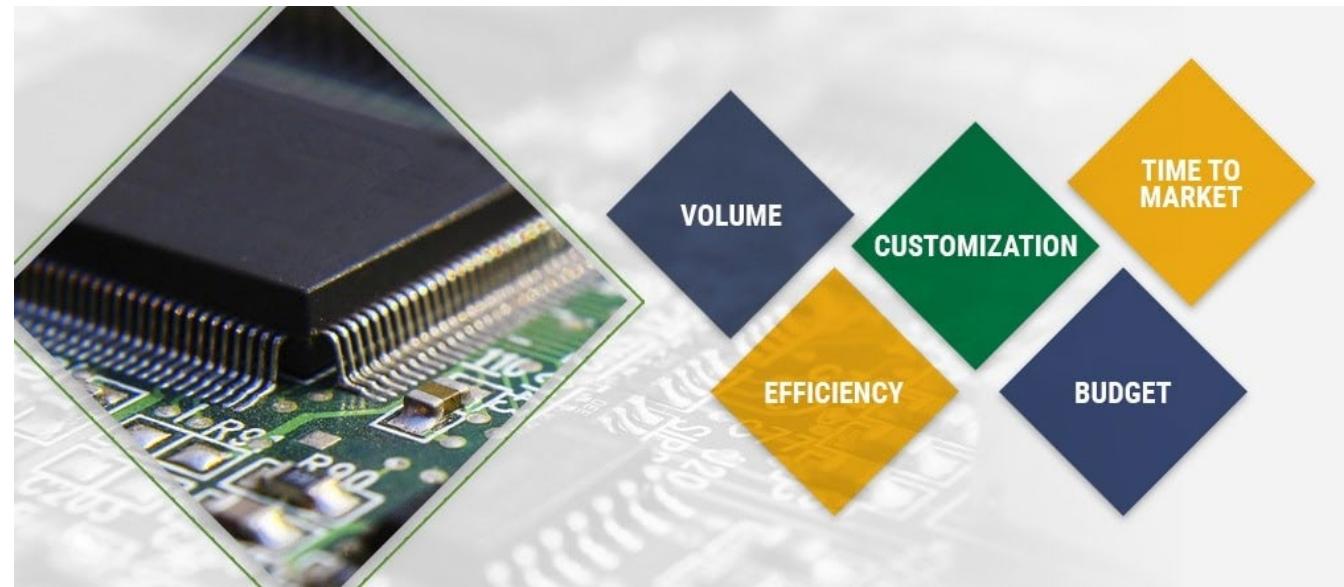
Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

Application-Specific Integrated Circuit (ASIC)

- An IC chip customized for a particular uses
- Cannot be reprogrammed for multiple applications
- Notably more efficiently than FPGAs
- Require a higher initial cost and longer design time compared to FPGA, more cost-effective if produced in large quantities
- Use: permanent applications in electronic devices
- NOTE: ASICs may be controlled with instructions



Google TPU v4



Languages and Compilers for Hardware Accelerators

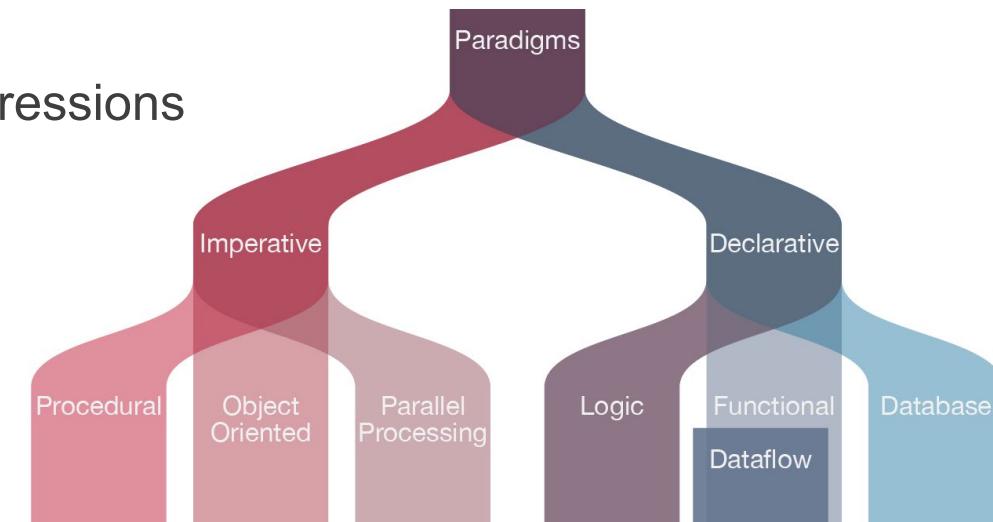
- **Programming languages:** formal language comprising a set of strings, used to implement algorithms
 - Provide an abstraction for underlying hardware
 - Provide a set of general operators to describe a range of applications
 - Primitive elements of programming languages
 - Syntax: rules that define the correct combination of symbols

Lisp
example:
(from
Wikipedia)

```
expression ::= atom | list
atom      ::= number | symbol
number    ::= [+ -]?['0'-'9']+
symbol    ::= ['A'-'Z' 'a'-'z'].*
list      ::= '(' expression* ')'
```

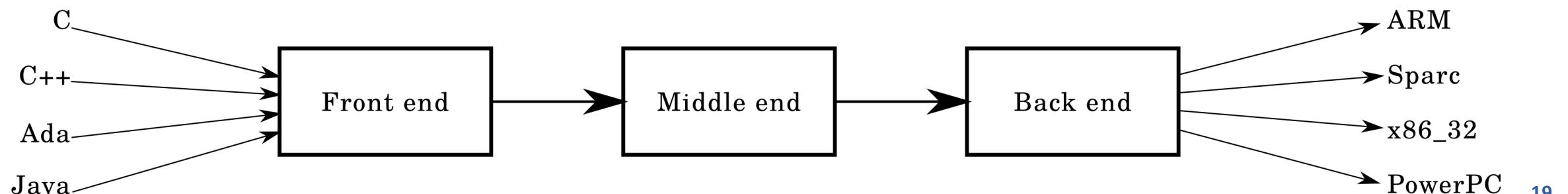
- an *expression* is either an *atom* or a *list*;
 - an *atom* is either a *number* or a *symbol*;
 - a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
 - a *symbol* is a letter followed by zero or more of any characters (excluding whitespace);
 - a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

- Semantics: meaning of languages
 - Type system: how a language classify values and expressions
 - Standard library and run-time system
 - Categories
 - Typed vs untyped languages, static vs dynamic typing
 - Functional programming vs imperative programming
 -



Languages and *Compilers* for Hardware Accelerators

- **Compilers:** a special program that processes code in a particular programming language and translates input code into the code in another or the same language.
- Categories
 - Just-In-Time (JIT) compiler, Ahead-of-Time (AOT) compiler
 - Source-to-source compiler
 -
- Three-stage compiler structure
 - Front end: translate source code to intermediate representation (IR), manages symbol table
 - Middle end: compiler analysis (e.g., data-flow analysis) and optimization (e.g., loop transformation)
 - Back end: architecture specific optimizations, code generation



Languages and Compilers for Hardware Accelerators

In this course, we will cover

- (I) languages and compilers that generate instructions or code that hardware accelerators can run,
 - Example: languages and compilers for an FPGA-based deep learning accelerator that takes specific instructions

Bits	63 – 60	59 – 56	55 – 32	31	30 – 24	23 – 17	16	15 – 10	9 – 4	3 – 0
input	Opcode = 0	Function	Destination Instruction ID	Fixed-Point/ Floating Point	Value Bitwidth	Fraction Bits/ Exponent Bits	1	# of Dimensions		
conv	Opcode = 1						Use the Next Word	Window Width	Window Height	Window Stride
pool	Opcode = 2							Window Width	Window Height	Window Stride
norm	Opcode = 3						# of Neurons			
ip	Opcode = 4									
act	Opcode = 5						0	Reserved		
fanout	Opcode = 6									
output	Opcode = 7									
		# Destinations		0						

Instructions of DNNWeaver accelerator
Source: “DNNWeaver: From High-Level Deep Network Models to FPGA Acceleration”

AND

- (II) languages and compilers that generate hardware accelerator designs
 - Example: languages and compilers that can generate an FPGA-based deep learning accelerator design
 - Multiple approaches: base on existing HLS flows (translate to C/C++), directly generates RTL from high-level language, OR, input language itself is a kind of HDL (hardware description language), etc.

Languages and Compilers for Hardware Accelerators

- *What?*
- *Why?*
- *How?*

Challenges in Modern Computing



60 seconds over the internet

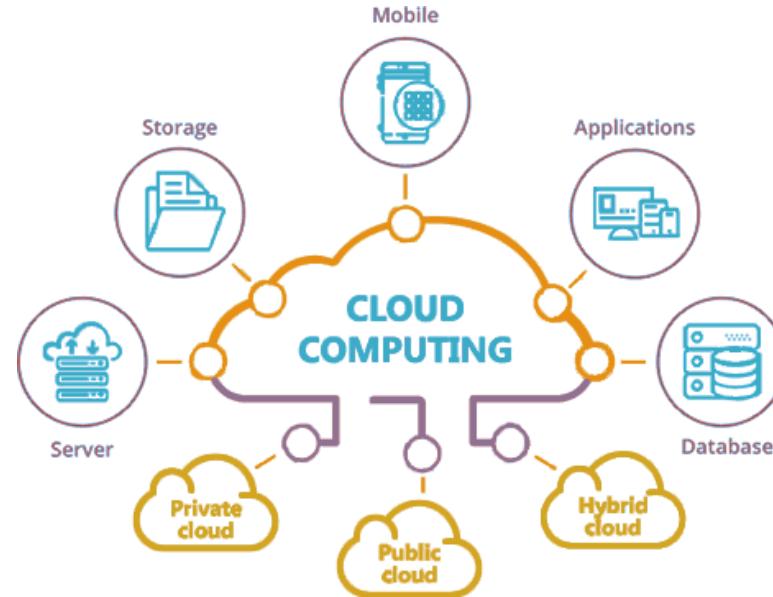
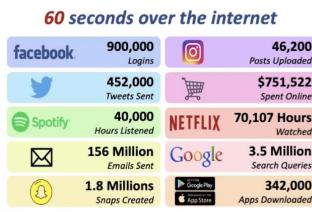


Complexity in Data

Challenges in Modern Computing



Complexity in Data



Cloud Computing



Edge Computing

Complexity in Applications

Challenges in Modern Computing



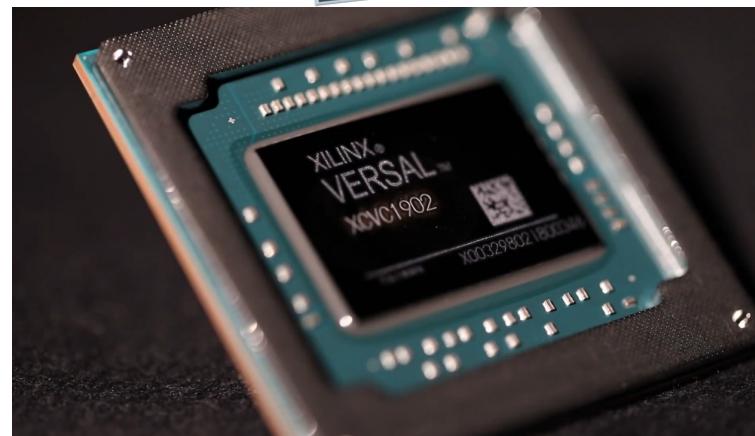
60 seconds over the internet

facebook	900,000 Logins	Instagram	46,200 Posts Uploaded
Twitter	452,000 Tweets Sent	Shopify	\$751,522 Spent Online
Spotify	40,000 Hours Listened	NETFLIX	70,107 Hours Watched
✉️	156 Million Emails Sent	Google	3.5 Million Search Queries
🔔	1.8 Millions Snaps Created	Google Play App Store	342,000 Apps Downloaded

Complexity in Data



Complexity in Applications



Complexity in Hardware



Challenges in Modern Computing



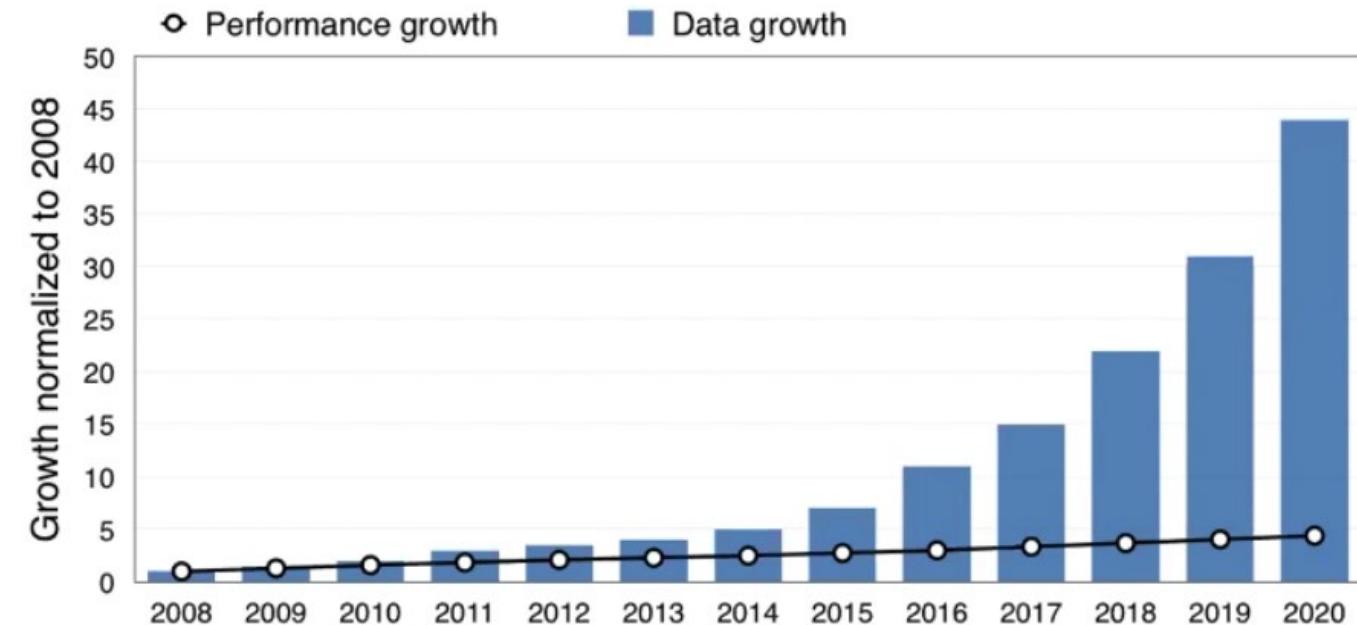
Complexity in Data



Complexity in Applications



Complexity in Hardware



Data growth trends: IDC's Digital Universe Study, December 2012

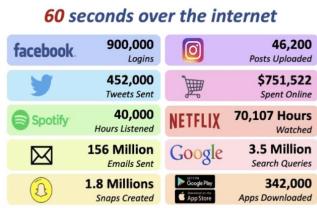
Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

Huge gap between data growth and processor performance growth!

Challenges in Modern Computing



Complexity in Data



Complexity in Applications



Complexity in Hardware

Accelerating modern applications with modern hardware is becoming more and more challenging!



Compilers automate the translation and optimization from application to hardware instructions

Challenges in Modern Computing

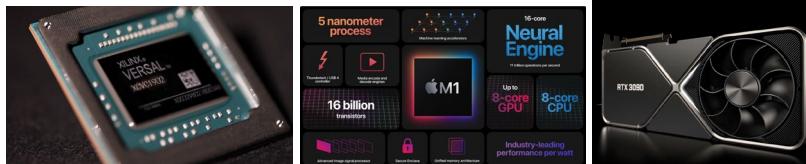


60 seconds over the internet	
facebook	900,000 Logins
Twitter	452,000 Tweets Sent
Spotify	40,000 Hours Listened
✉️	156 Million Emails Sent
🔔	1.8 Millions Snaps Created
Instagram	46,200 Posts Uploaded
🛒	\$751,522 Spent Online
NETFLIX	70,107 Hours Watched
Google	3.5 Million Search Queries
apk	342,000 Apps Downloaded

Complexity in Data



Complexity in Applications



Complexity in Hardware



Data growth trends: IDC's Digital Universe Study, December 2012

Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011

Challenges in Modern Computing



Complexity in Data

60 seconds over the internet	
facebook	900,000 Logins
Instagram	46,200 Posts Uploaded
Twitter	452,000 Tweets Sent
Spotify	\$751,522 Spent Online
NETFLIX	40,000 Hours Listened
EMAIL	70,107 Hours Watched
Google	156 Million Emails Sent
GOOGLE PLAY	3.5 Million Search Queries
Facebook	1.8 Millions Snaps Created
APP STORE	342,000 Apps Downloaded



Complexity in Applications



Complexity in Hardware



Jeff Dean. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design, arXiv 1911.05289.

Challenges in Modern Computing



60 seconds over the internet	
facebook	900,000 Logins
Twitter	452,000 Tweets Sent
Spotify	40,000 Hours Listened
	NETFLIX 70,107 Hours Watched
	Google 3.5 Million Search Queries
	156 Million Emails Sent
	1.8 Millions Snaps Created
	342,000 Apps Downloaded

Complexity in Data



Complexity in Applications



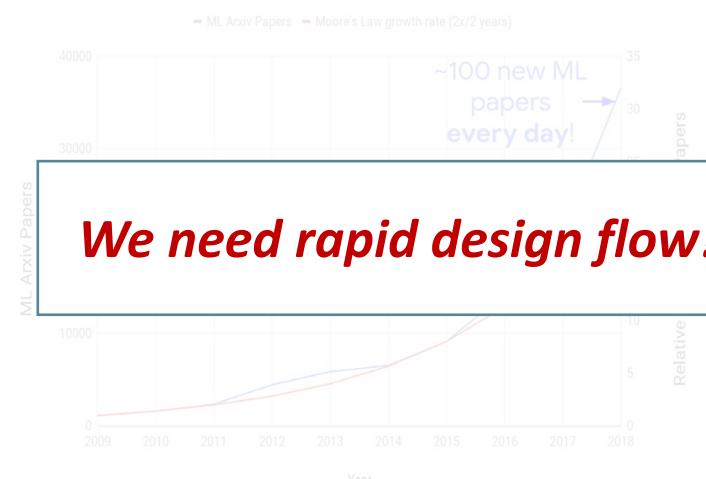
Complexity in Hardware



We need efficient acceleration systems!

Data growth trends: IDC's Digital Universe Study, December 2012

Performance growth trends: Hadi Esmaeilzadeh et al, "Dark Silicon and the End of Multicore Scaling", ISCA 2011



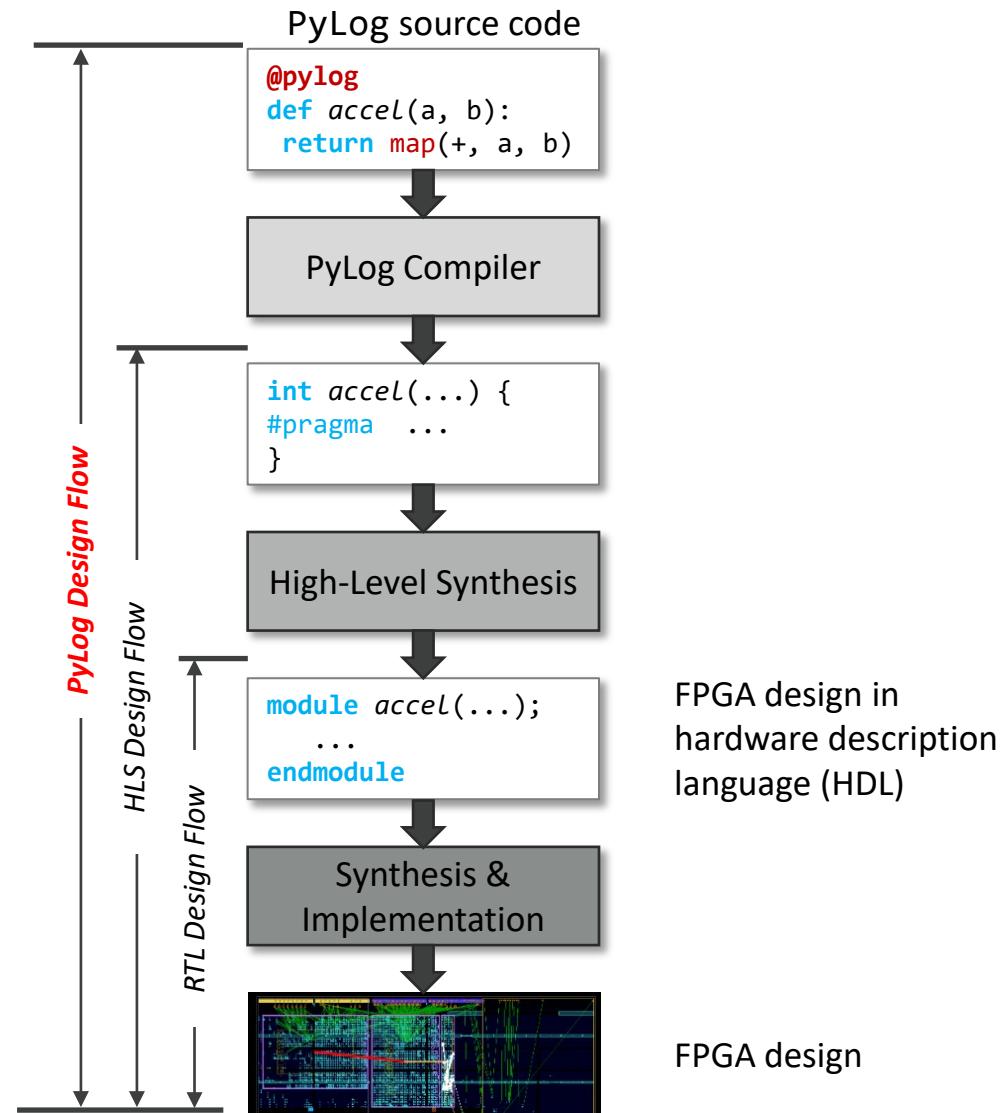
We need rapid design flow!

Jeff Dean. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design, arXiv 1911.05289.

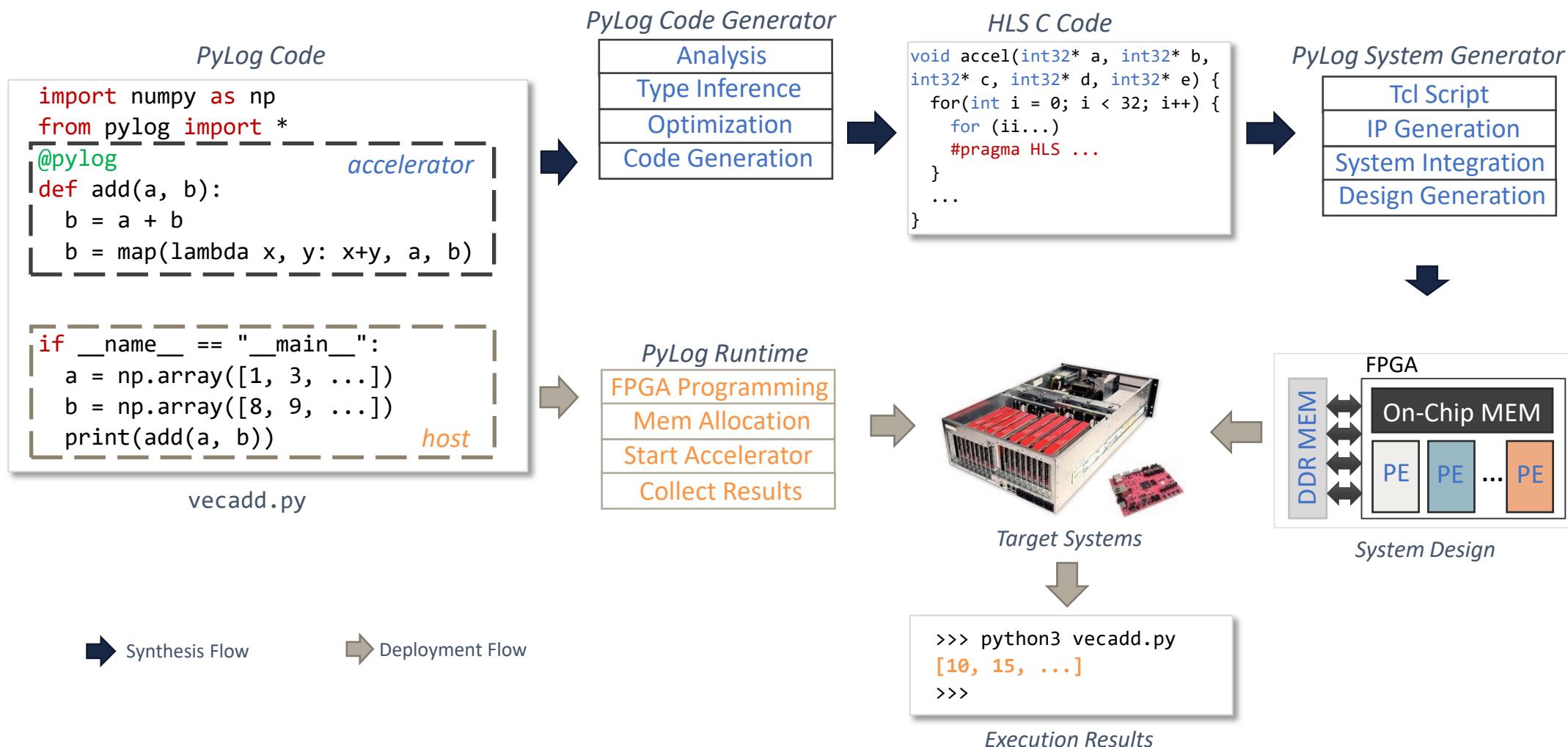
Languages and Compilers for Hardware Accelerators

- *What?*
- *Why?*
- *How?*

Example 1: FPGA-based Accelerators, HLS, and PyLog



Example 1: FPGA-based Accelerators, HLS, and PyLog

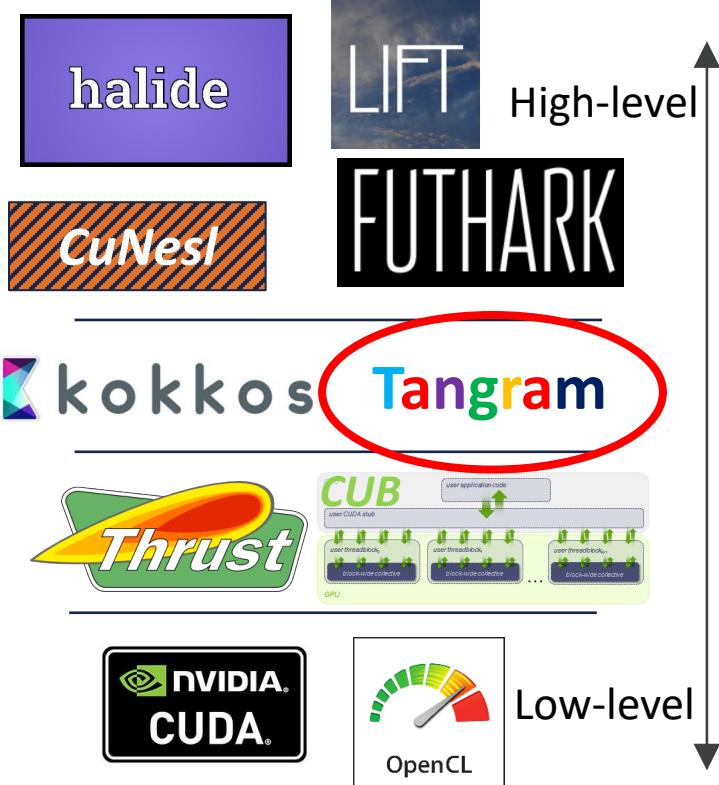




Example 2: **Tangram**: Efficient GPU Code Generation

Goal: performance portable GPU code generation (for different GPU generations, different applications, etc.)

GPU programming strategies:



```
_codelet
int sum(const Array<1,int> in) {
    unsigned len = in.size();
    int accum = 0;
    for(unsigned i=0; i < len; ++i) {
        accum += in[i];
    }
    return accum;
}
```

(a) Atomic autonomous codelet

```
_codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
    __shared int tmp[coopDim()];
    unsigned len = in.size();
    unsigned id = coopIdx();
    tmp[id] = (id < len)? in[id] : 0;
    for(unsigned s=1; s<coopDim(); s *= 2) {
        if(id >= s)
            tmp[id] += tmp[id - s];
    }
    return tmp[coopDim()-1];
}
```

(b) Atomic cooperative codelet

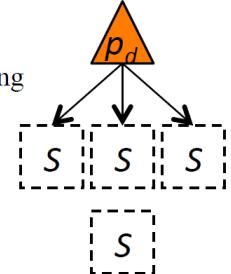
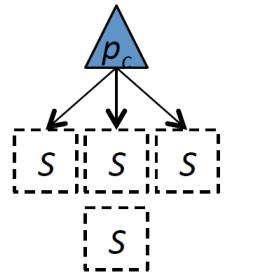


```
_codelet __tag(asso_tiled)
int sum(const Array<1,int> in) {
    __tunable unsigned p;
    unsigned len = in.size();
    unsigned tile = (len+p-1)/p;
    return sum( map( sum, partition(in,
        p,sequence(0,tile,len),sequence(1),sequence(tile,tile,len+1))));
}
```

(c) Compound codelet using adjacent tiling

```
_codelet __tag(stride_tiled)
int sum(const Array<1,int> in) {
    __tunable unsigned p;
    unsigned len = in.size();
    unsigned tile = (len+p-1)/p;
    return sum( map( sum, partition(in,
        p,sequence(0,1,p),sequence(p),sequence((p-1)*tile,1,len+1))));
}
```

(d) Compound codelet using strided tiling





Example 2: **Tangram**: Efficient GPU Code Generation

Goal: performance portable GPU code generation (for different GPU generations, different applications, etc.)

GPU programming strategies:

halide

LIFT

High-level

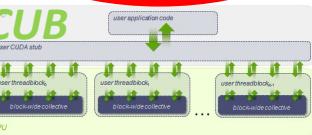
CuNesi

FUTHARK

kokkos

Tangram

Thrust



NVIDIA CUDA

OpenCL

Low-level

Spectrum represents a unique computation with a defined set of inputs and outputs
Codelet represents a specific algorithmic implementation of a spectrum
A spectrum can have many codelets (versions), e.g., sequential, distribute, etc.

Tangram Composition Rules

Select(S_1, L_i) → Compose ($c_x \in S_1, L_i$);

Compose(c_x, L_i) → Devolve (c_x, L_i);
→ Distribute (c_m, L_i);
→ Compute ($c_{s/v}, L_i$);

Devolve(c_x, L_i) → Compose(c_x, L_{i-1});

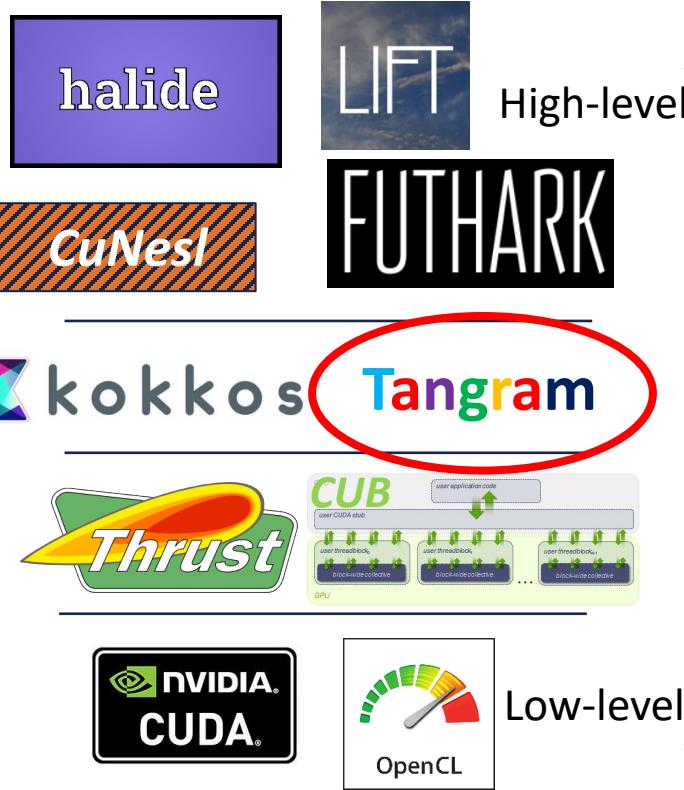
Distribute(c_m, L_i) → Regroup([Compose(c_x, L_{i-1})₁, ..., Compose(c_x, L_{i-1})_P] , L_i);

Regroup(P, L_i) → Select (S_2, L_i);

Example 2: **Tangram**: Efficient GPU Code Generation

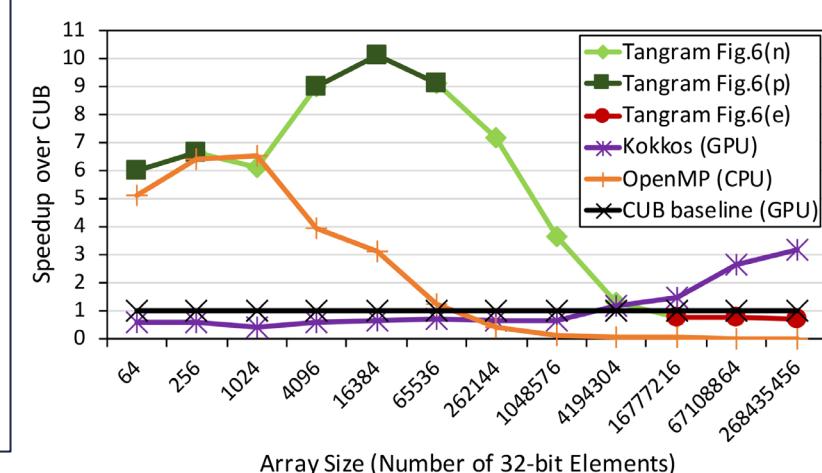
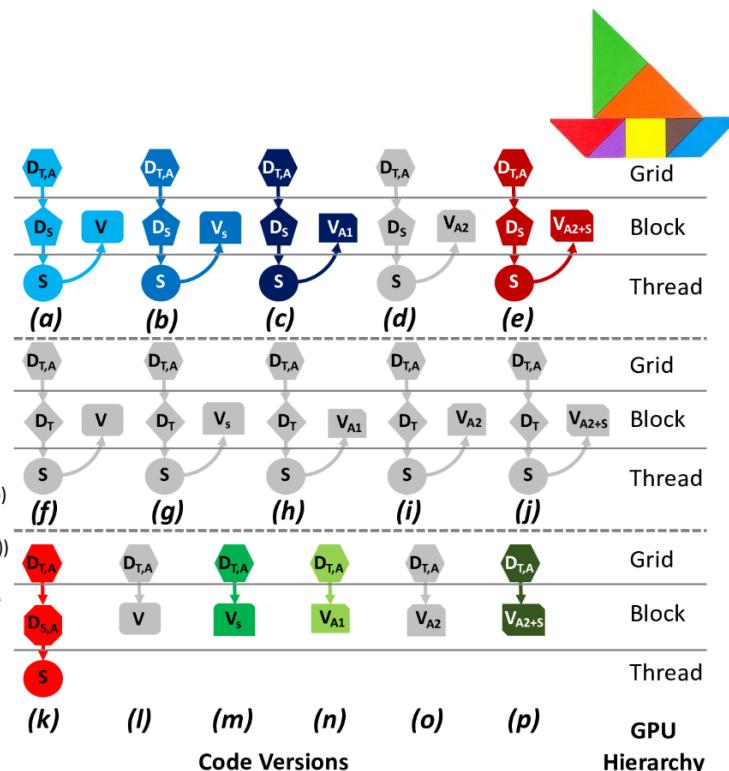
Goal: performance portable GPU code generation (for different GPU generations, different applications, etc.)

GPU programming strategies:



Spectrum represents a unique computation with a defined set of inputs and outputs
Codelet represents a specific algorithmic implementation of a spectrum
A spectrum can have many codelets (versions), e.g., sequential, distribute, etc.

- D_T Tile Distribute (Figure 1(b))
- D_S Stride Distribute (Figure 1(b))
- $D_{T,A}$ Global Atomic Tile Distribute
- $D_{S,A}$ Global Atomic Stride Distribute
- V Cooperative (Figure 1(c))
- V_s Cooperative + Shuffle
- V_{A1} Shared Memory Atomic 1 (Figure 3(a))
- V_{A2} Shared Memory Atomic 2 (Figure 3(b))
- V_{A2+s} Shared Memory Atomic 2 + Shuffle
- S Scalar (Figure 1(a))



Tangram Composition Rules		Codelets and Variants
Select(S_1 , L_i)	\rightarrow	Compose ($c_x \in S_1, L_i$);
Compose(c_x, L_i)	\rightarrow	Devolve (c_x, L_i); \rightarrow Distribute (c_m, L_i); \rightarrow Compute ($c_{s/v}, L_i$);
Devolve(c_x, L_i)	\rightarrow	Compose(c_x, L_{i-1});
Distribute(c_m, L_i)	\rightarrow	Regroup([Compose(c_x, L_{i-1}) ₁ , ..., Compose(c_x, L_{i-1}) _P] , L_i);
Regroup(P, L_i)	\rightarrow	Select (S_2, L_i);

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 2:

Hardware Accelerators

Sitao Huang

sitaoh@uci.edu

January 11, 2022



Tentative Schedule

- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compiler Optimizations for Accelerators
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

Hardware Accelerators

- **Hardware accelerator:** “computer hardware designed to perform specific functions more efficiently compared to software running on a CPU” (Wikipedia)
- Hardware accelerators have been there since early days
- Intel 8087: the first x87 floating-point coprocessor for the 8086 line of microprocessors (announced in 1980)
 - Speed up computations for floating-point arithmetic
- Digital Signal Processor (DSP)
 - Accelerates digital signal processing
- Graphics Processing Unit (GPU)
 - Specialized for graphics processing, now also used for general computing
- FPGA: reconfigurable domain-specific accelerators
- ASIC: customized accelerators

A Taxonomy of Accelerators* *(from host processor's perspective)*

- Granularity: what kinds of computation are offloaded to accelerator?
 - Instruction level: primitives like arithmetic operators, e.g., sqrt, sin/cos, etc.
 - Kernel level: functional units, modules, e.g., matrix multiply, FFT, etc.
 - Application level: accelerates entire applications, e.g., DNN inference, video decoding, etc.
 - Coupling (with host): where accelerators are deployed in the system?
(assumed system hierarchy: pipelined processor core with multiple levels of caches attached to the memory bus and then connected to I/O devices through I/O bus)
 - Part of the processor pipeline
 - Attached to cache
 - Attached to memory bus
 - Attached to the I/O bus
- 
- Tightly coupled with host processor
 - more design constraints, lower invocation overhead
 - Loosely coupled with host processor
 - less design constraints, higher invocation overhead

*Source: Yakun Sophia Shao and David Brooks, Research Infrastructures for Hardware Accelerators, 2015

A Taxonomy of Accelerators* *(from host processor's perspective)*

	Part of the Pipeline	Attached to Cache	Attached to the Memory Bus	Attached to the I/O Bus
Instruction-Level	FPU, SIMD, DySER [58],	Hwacha [76, 95, 121], CHARM [43, 44],		
Kernel-Level	NPU [52], 10x10 [40], Convolution Engine [98], H.264 [61],	SNNAP [91], C-Cores [119],	Database [35], Q100 [126], LINQits [41], AccStore [86],	
Application-Level	x86 AES [18], Oracle/Cavium Crypto Acc [11, 69],	Key-Value Stores [87], Memcached [80],	Sonic3D [103], DianNao [27, 38], HARP [125], TI OMAP5 [16], IBM PowerEN [71], IBM POWER7+ [30],	GPU, Catapult [97], IBM Power8 CAPI Acc [4],

*Source: Yakun Sophia Shao and David Brooks, Research Infrastructures for Hardware Accelerators, 2015

Accelerator Design

- What to accelerate?
 - Decide the operational specifications of the hardware accelerator
 - Profile software applications
 - Determine the critical path/bottleneck, and frequently used kernels or functions
- How to accelerate?
 - Architecture of the accelerator
 - Memory hierarchy and I/O interfaces
 - CPU-accelerator interfaces
 - Programming interfaces
- Acceleration goals/requirements/constraints?
 - Maximum latency
 - Minimum throughput
 - Maximum power consumption
 - Cost, time to market, etc.

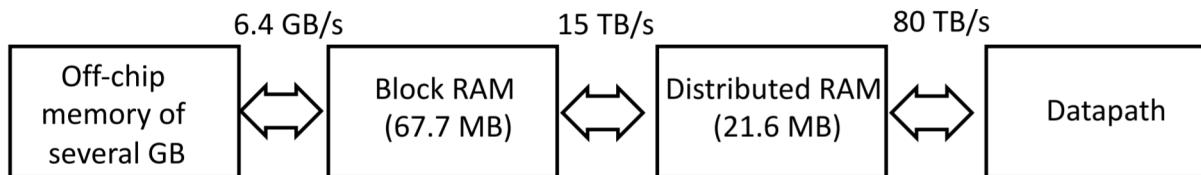
Accelerator Design

A few examples of choices in hardware accelerator design

- Types of parallelism exploited
 - Fine-grained vs coarse-grained
 - Data parallel vs task parallel
- Optimized for high throughput vs low latency
 - E.g., optimizing number of tasks completed per unit of time, OR, execution time of a single task
- Memory organization
- External interfaces
- On-chip memory usage, data buffering schemes

Basic Concepts

- Latency
 - Time required to perform certain task or to produce certain result.
 - Measured in units of time, e.g., hours, minutes, seconds, nanoseconds, or clock cycles
 - Applications that need low latency: object detection/tracking, DNN inference, etc.
- Throughput
 - The number of tasks or results produced per unit of time
 - Memory bandwidth: how much data is moved through memory interface per unit time. MB/s or GB/s
 - Computational throughput: how much computation is done per unit time. FLOP/s, OP/s, IOP/s
 - Applications that need high throughput: image/video post-processing, DNN training, etc.

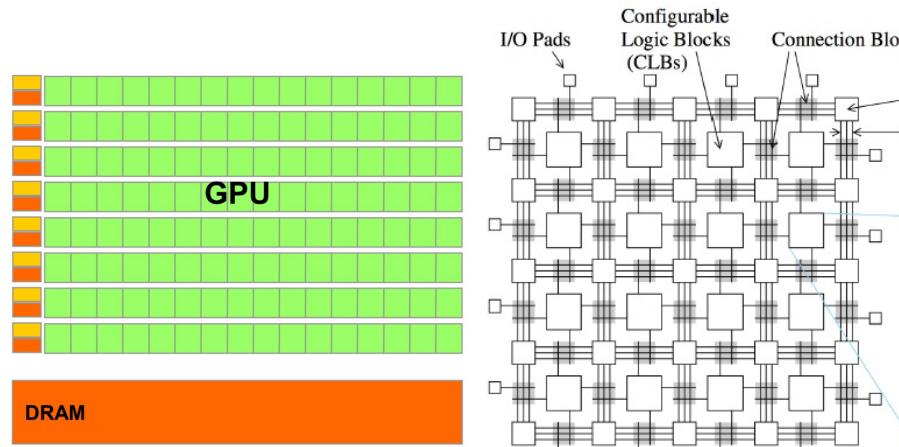


Bandwidth/memory distribution in Xilinx Virtex-7 FPGA

(source: F. Siddiqui et al, "FPGA-Based Processor Acceleration for Image Processing Applications")

Parallelism

- Why are accelerators faster?
 - Exploit the parallelism in kernels/applications
- Types of parallelism
 - Fine-grained (low level) vs coarse-grained (high level)
 - Instruction level
 - Thread level
 - Task level
 - Data level
 - ... (name your levels)
 - Data parallel vs task parallel



Parallel Hardware Design

- Consider vector addition:

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

- No data dependences between loop iterations
- Explicit data parallelism in this example
- We could instantiate K parallel adders
 - Speedup = N/K
 - *Can we really achieve N/K speedup?*

Parallel Hardware Design

- Parallel processing units come with a cost
 - More area
 - More power consumption
 - Higher complexity in place & route (could lead to worse timing)
- In our vector addition example
 - In each loop iteration: 2 reads and 1 write for 1 add
 - Assume all values are 32-bit floating-point numbers, that requires reading 8 bytes of data per add
- *How much memory bandwidth we need for supplying input data to K-wide adder? (not considering writes)*
 - $8*K$ bytes/s

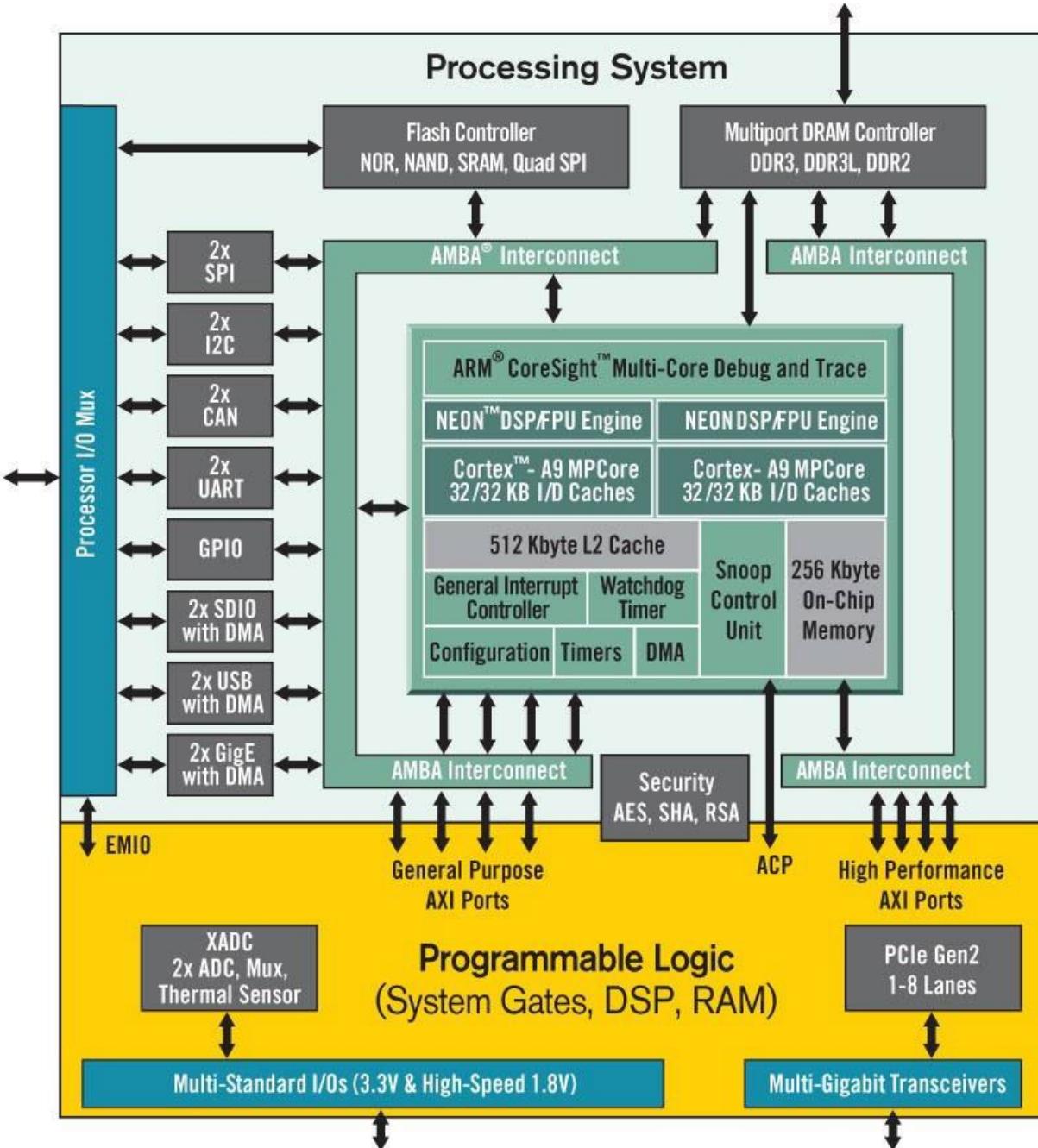
```
for (i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

Parallel Hardware Design

- For a specific platform, an application can be *Compute Bound* or *Memory Bound*
 - “Compute” to “Memory” ratio:
$$\frac{\text{Number of operations (FLOPs)}}{\text{Data transferred through memory for the operations (Bytes)}}$$
- Understand the nature of the application and optimize the design accordingly
- Some design techniques can be used to change the “Compute” to “Memory” ratio
 - Example 1: increase data reuse rate using on-chip memory (increase the ratio)
 - Example 2: re-compute intermediate results without write backs (increase the ratio)
 - Example 3: Write back intermediate results immediately (save on-chip memory, decrease the ratio)

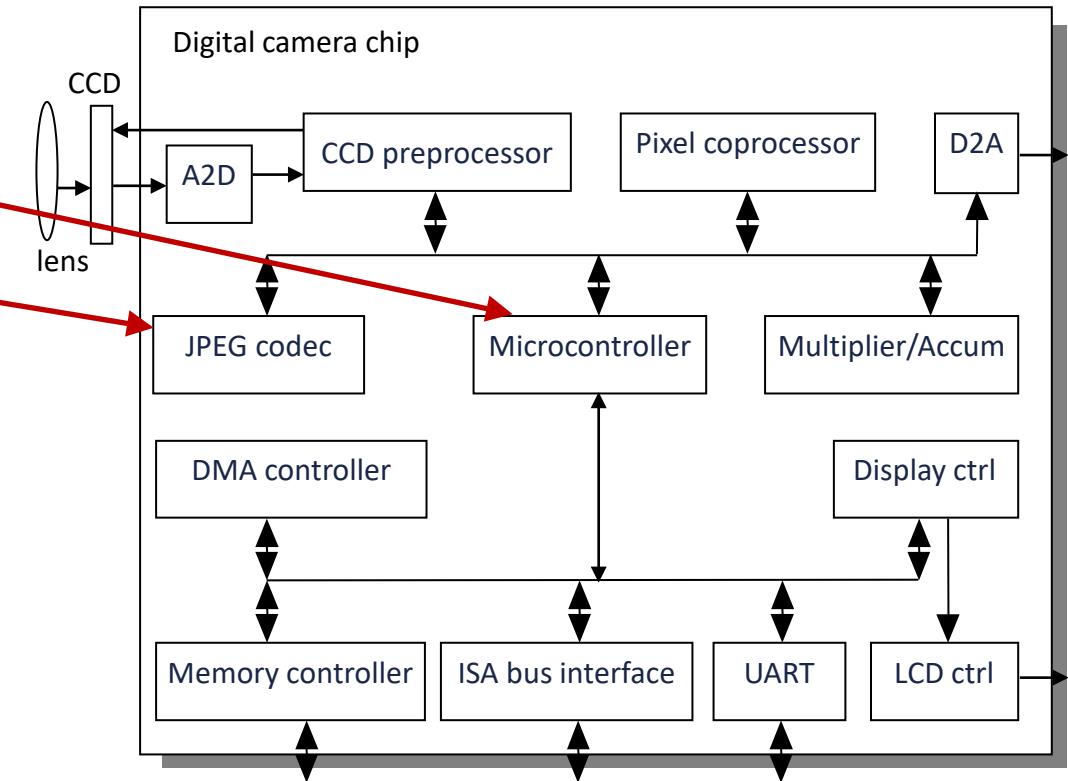
Interface Choices

- How do data move in and out of the accelerator?
- What are the bandwidths needed for the interfaces?

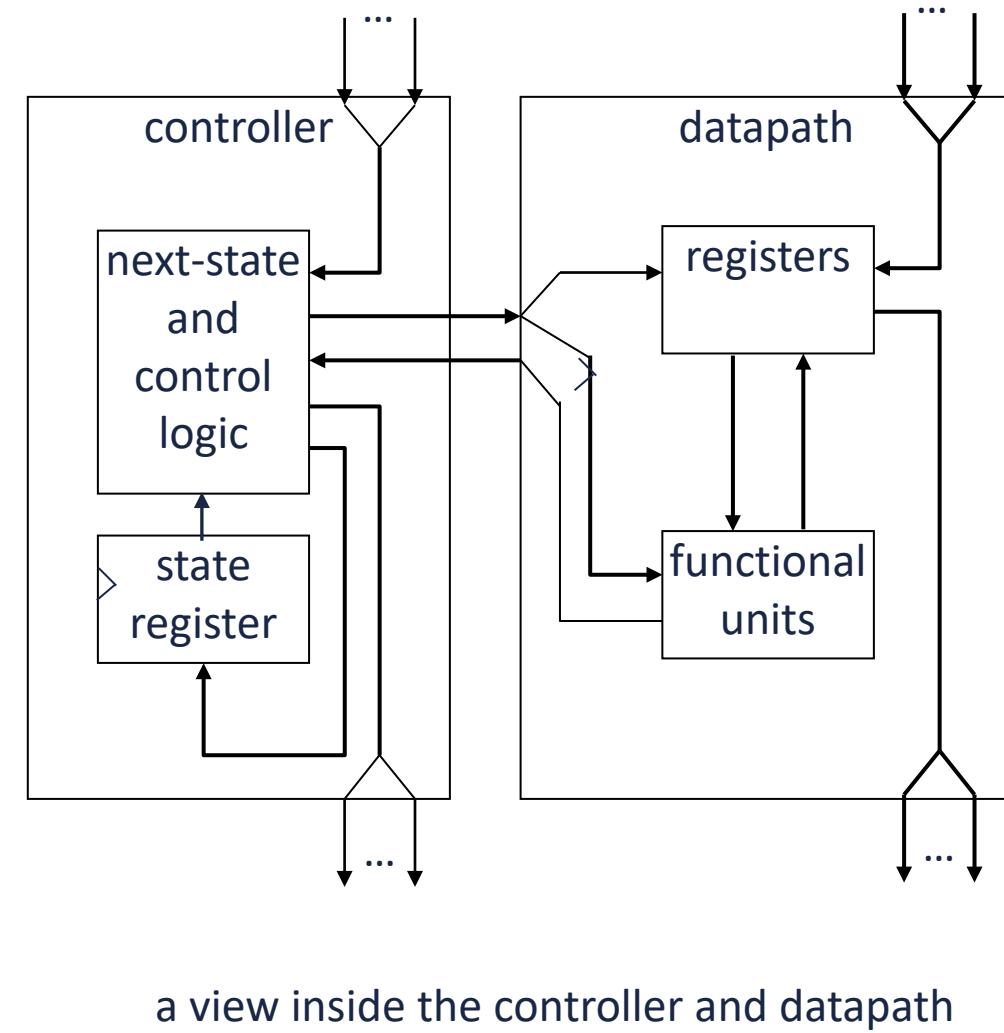
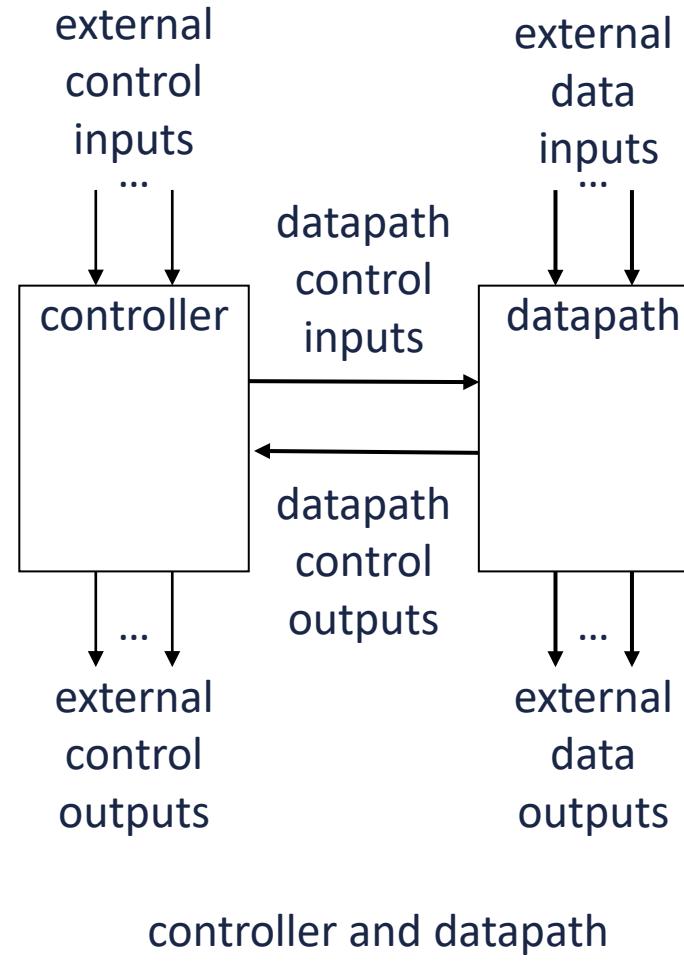


Designing Single-Purpose Processors

- Processor
 - Digital circuit that performs computation tasks
 - Contains controller and datapath
 - General-purpose: variety of computation tasks
 - Single-purpose: one particular computation task
 - Application-specific instruction-set processor (ASIP): domain specific tasks
- A custom single-purpose processor may be
 - Fast, small, low power
 - But, high NRE (Non-Recurring Engineering) cost, longer time-to-market, less flexible



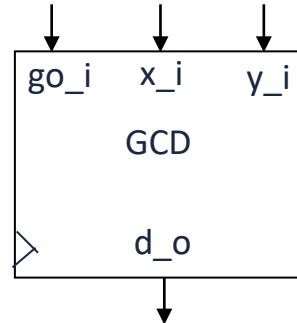
Custom Single-Purpose Processor Basic Model



Example: Greatest Common Divisor (GCD)

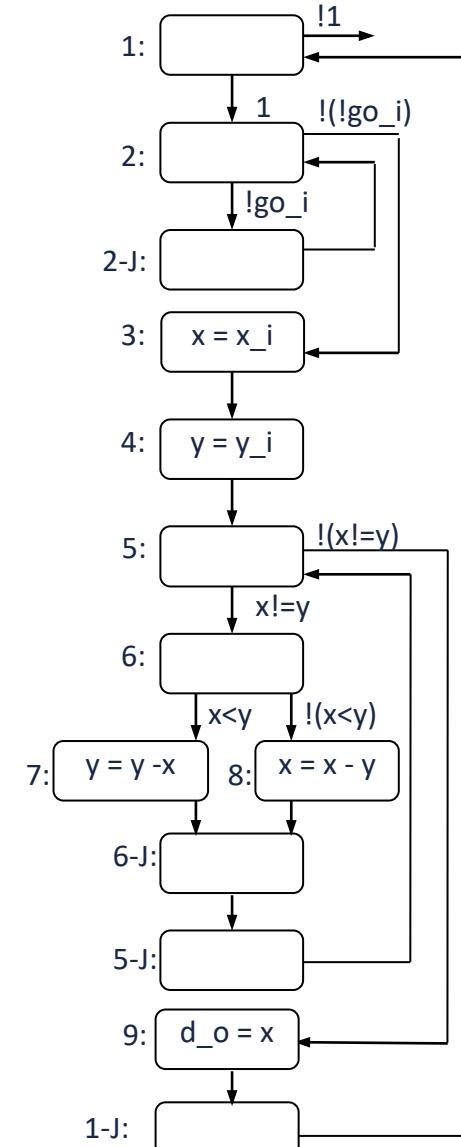
- First, create algorithm
- Then, convert algorithm to “complex” state machine
 - Known as FSMD: Finite-State Machine with Datapath
 - Can use templates to perform such conversion

(a) black-box view



```
0: int x, y;  
1: while (1) {  
2:   while (!go_i);  
3:   x = x_i;  
4:   y = y_i;  
5:   while (x != y) {  
6:     if (x < y)  
7:       y = y - x;  
     else  
8:       x = x - y;  
   }  
9:   d_o = x;  
}
```

(b) desired functionality

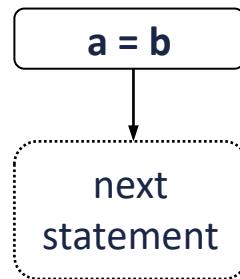


(c) FSMD

State Diagram Templates

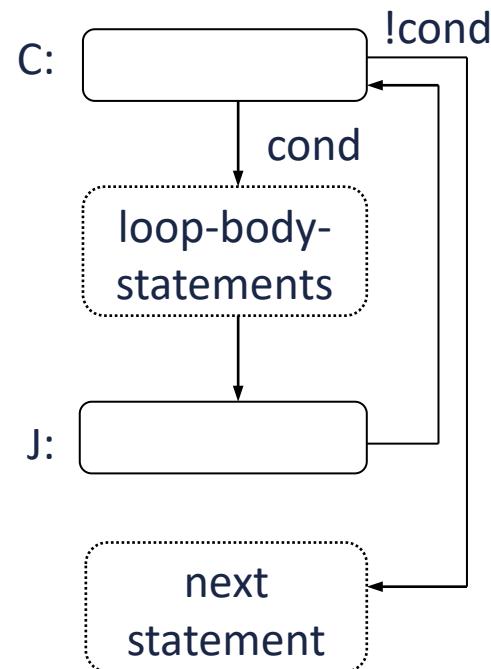
Assignment statement

a = b
next statement



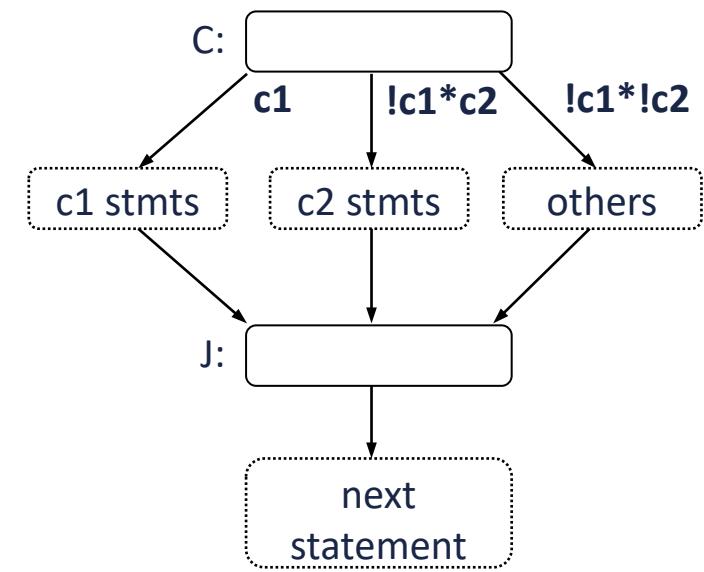
Loop statement

while (cond) {
loop-body-
statements
}
next statement



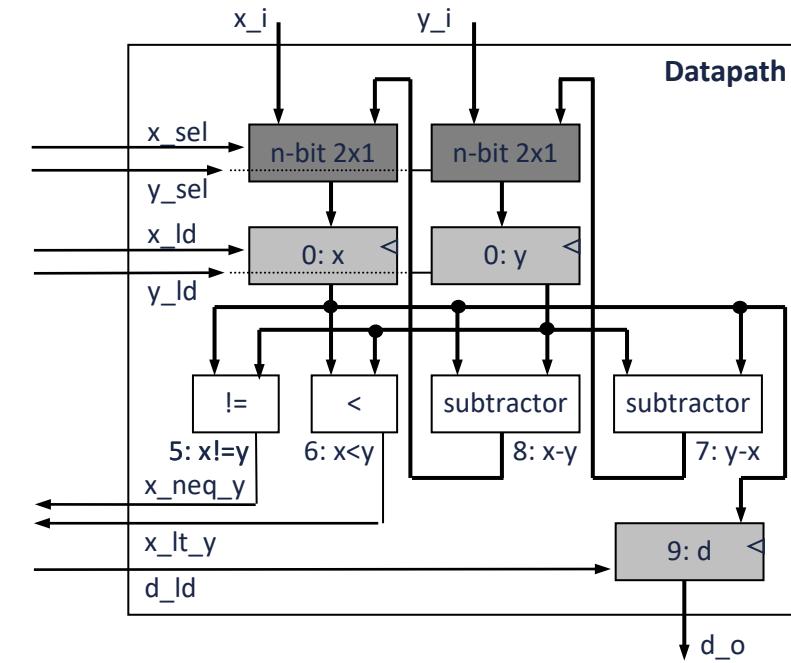
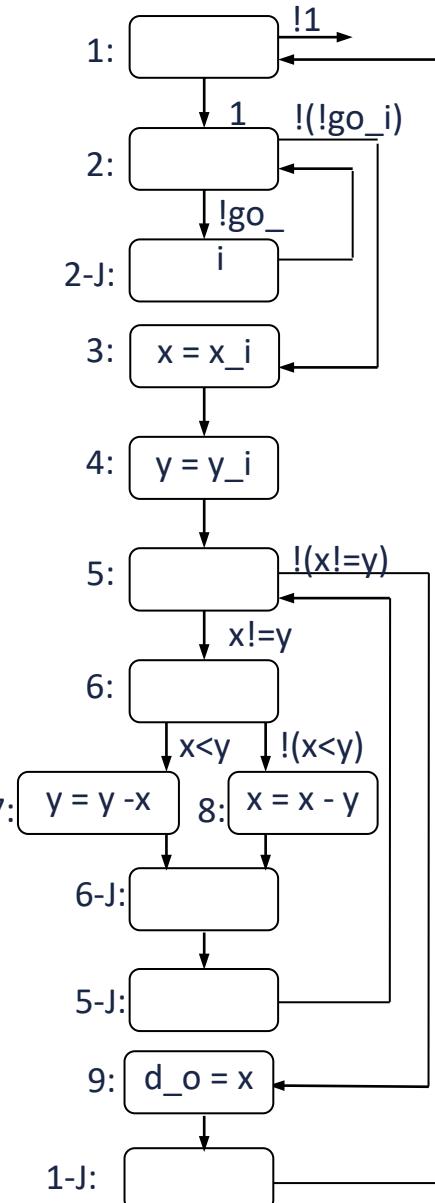
Branch statement

if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement

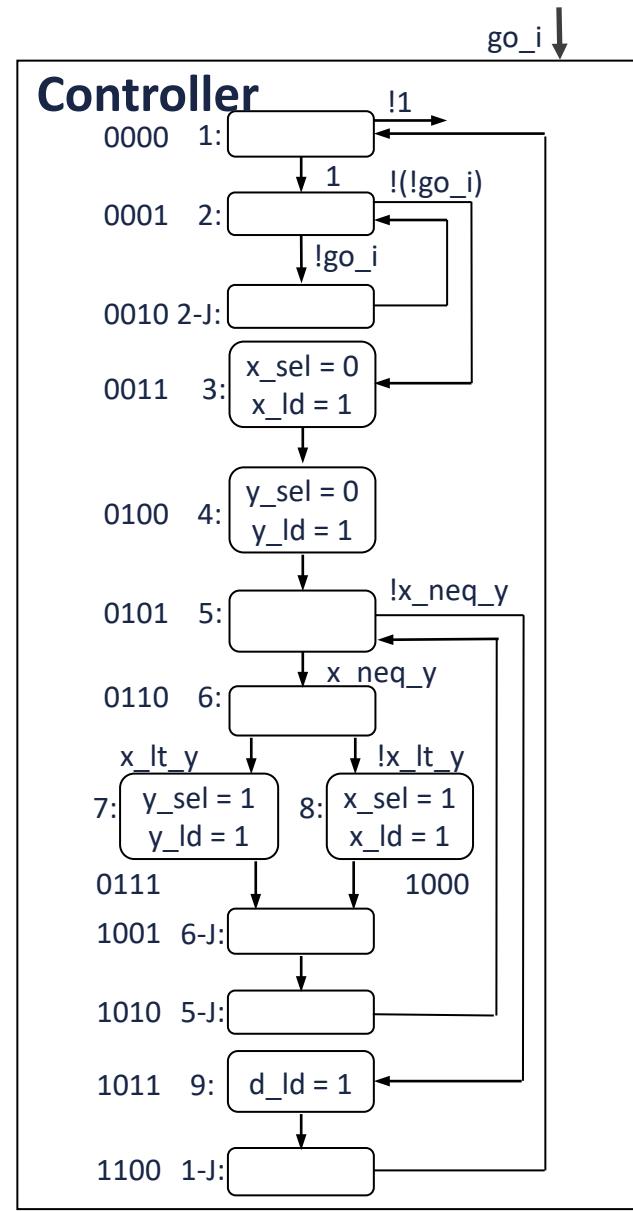
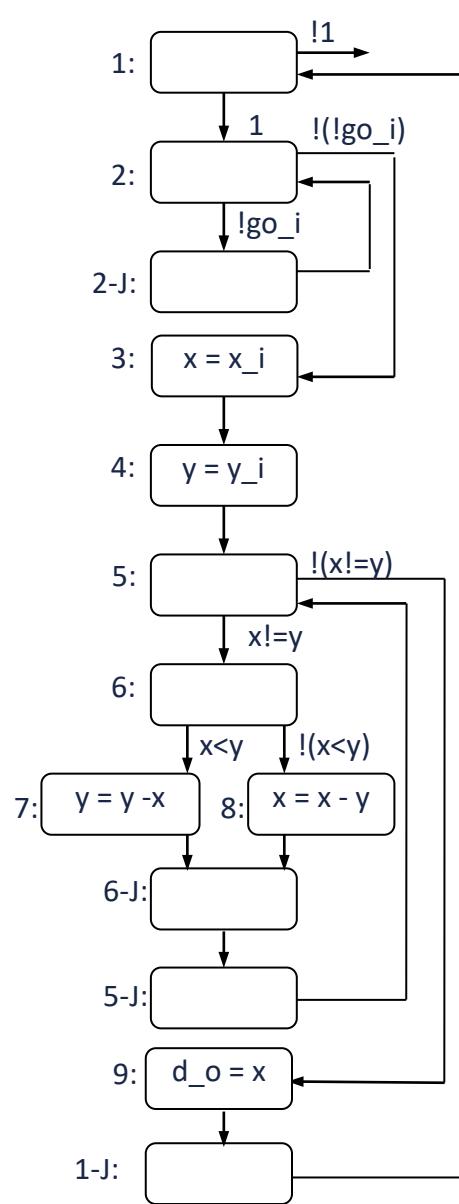


Creating the Datapath

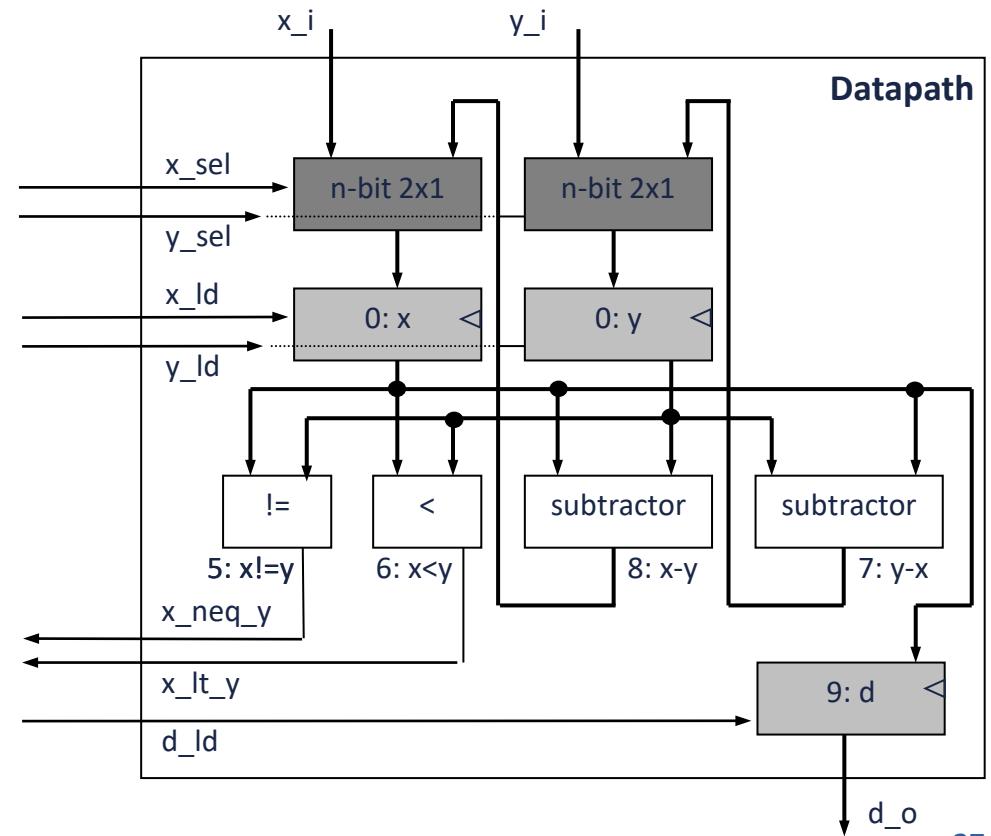
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers, and functional units
 - Based on reads and writes
 - Use multiplexors for multiple sources
- Create unique identifier
 - For each datapath component control input and output



Creating the Controller's FSM

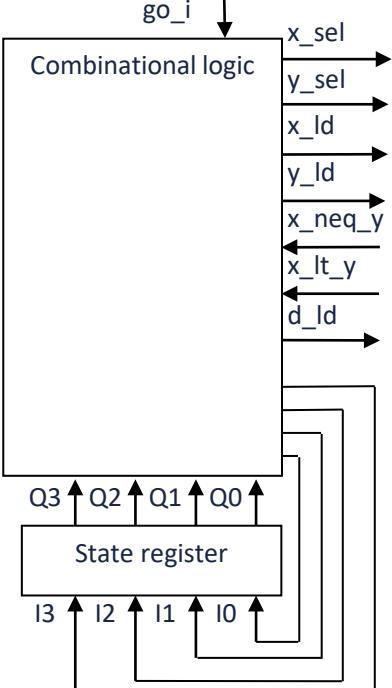


- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations

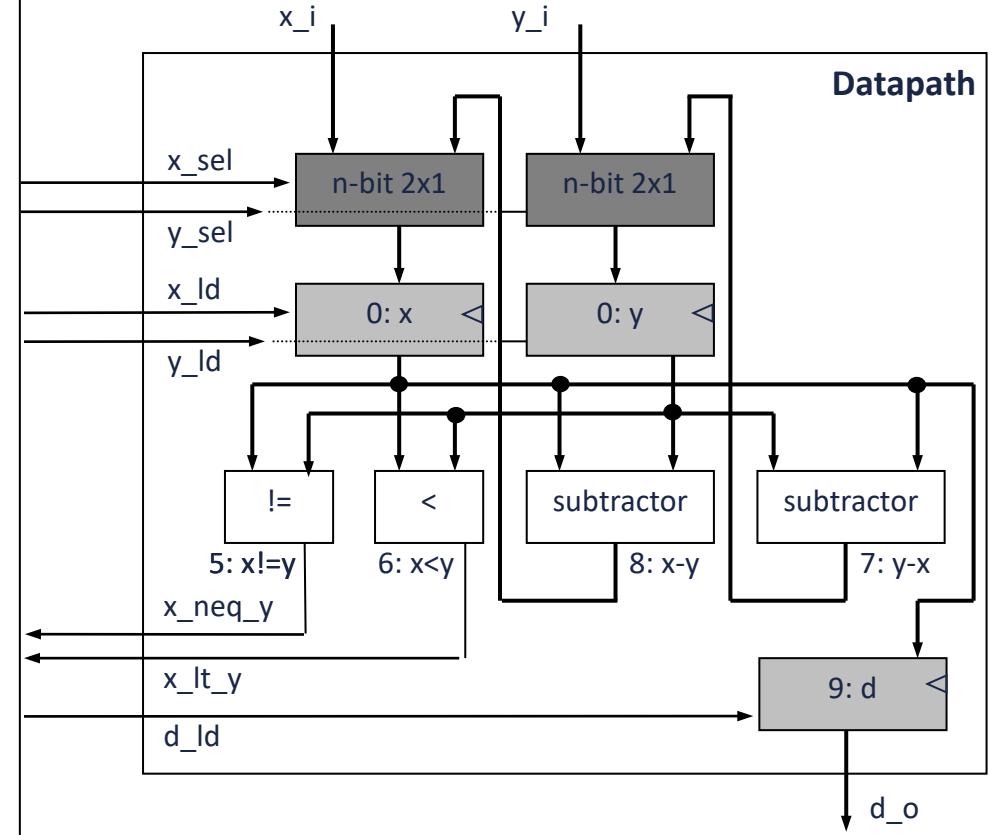
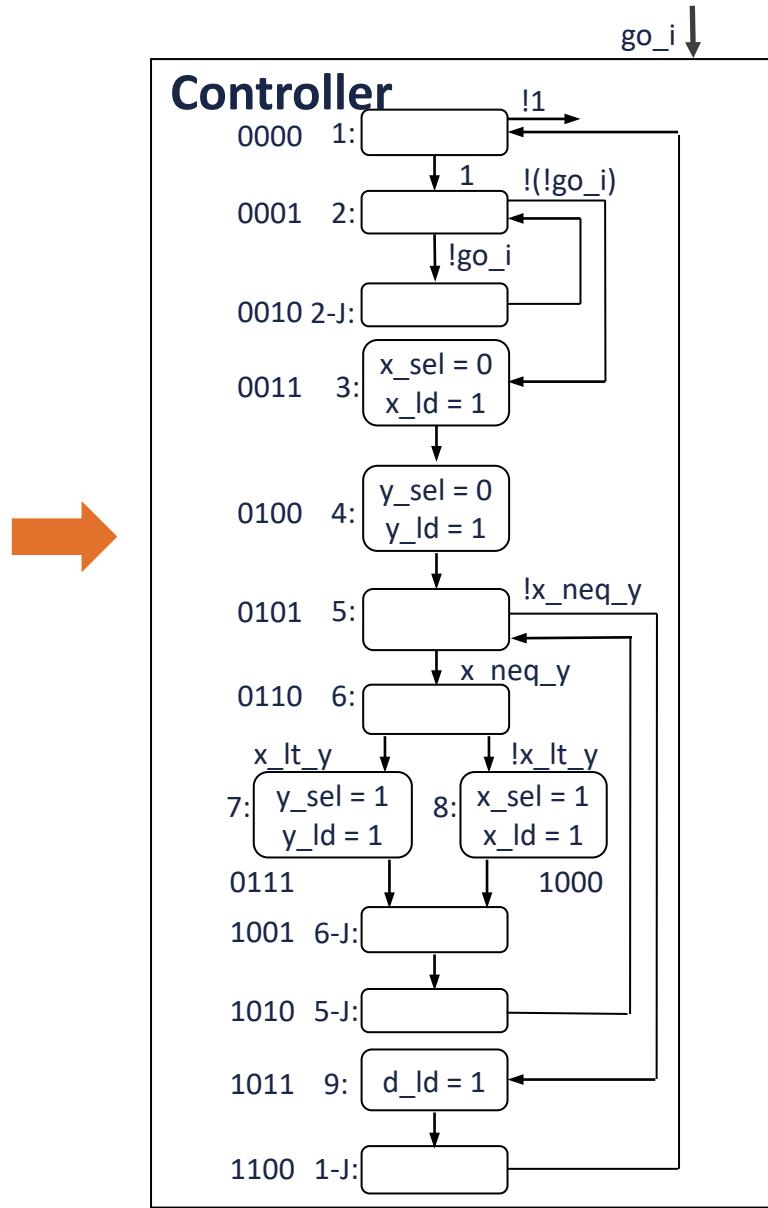


Splitting into a Controller and Datapath

Controller implementation model

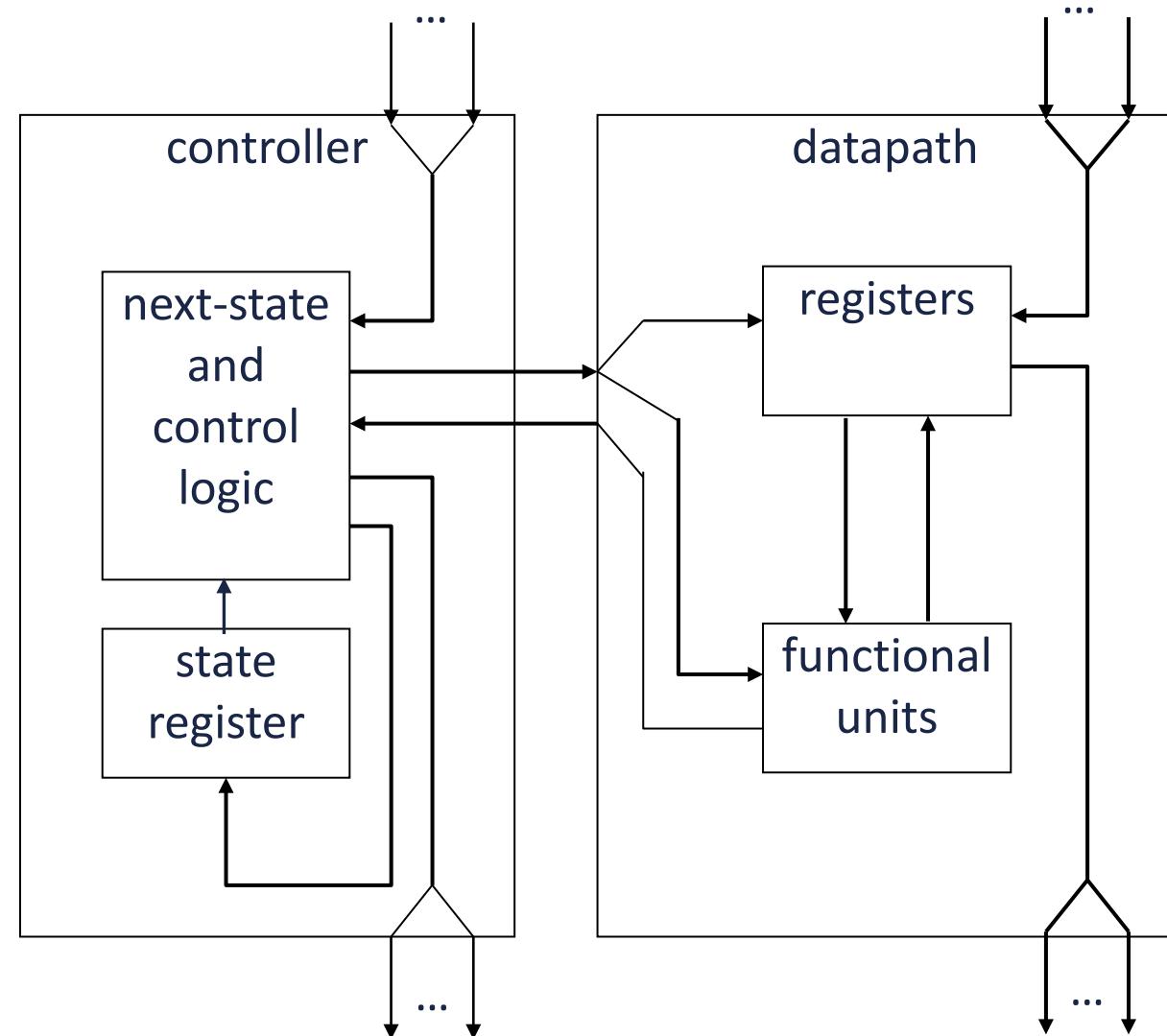


Block View of
the Controller



Completing the GCD Single-Purpose Processor Design

- Next Steps
 - Create state table for the next state and control logic
 - Combinational logic design
 - Optimize the processor design
- Next, we will show how to *optimize this processor design*



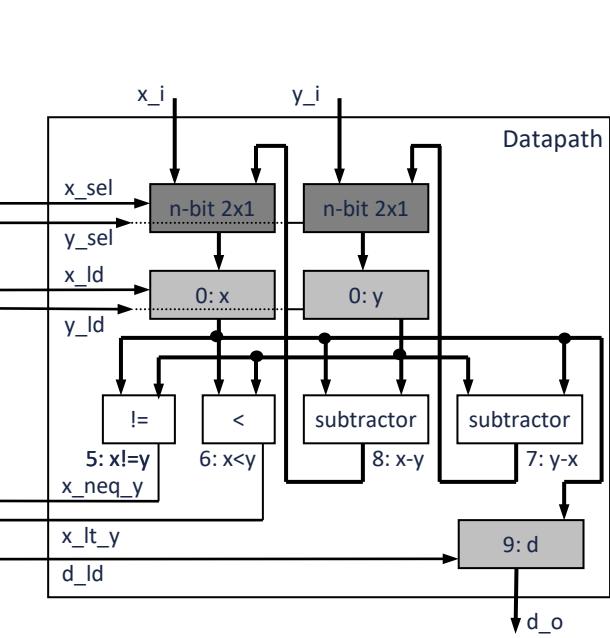
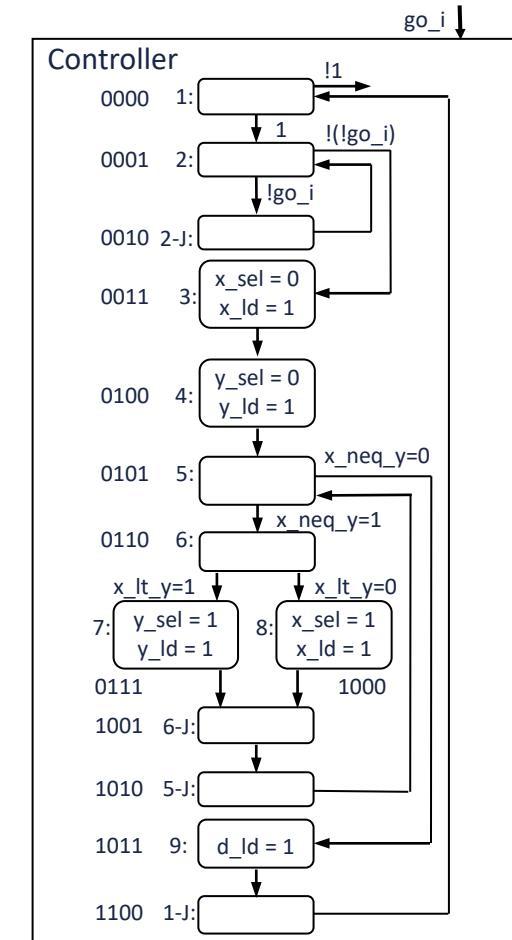
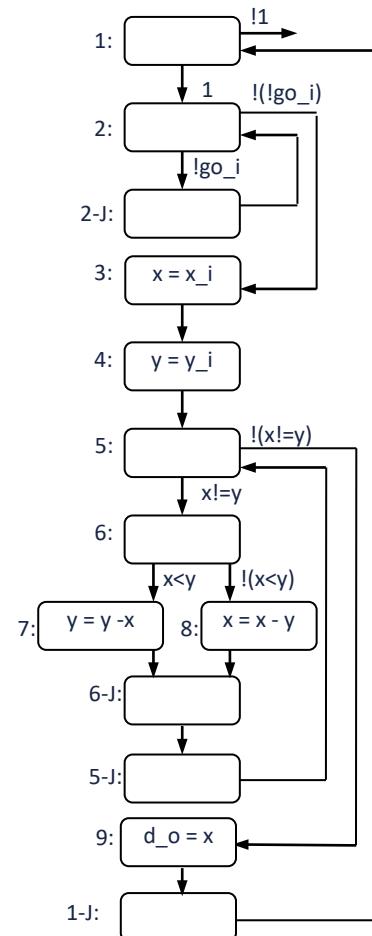
a view inside the controller and datapath

Optimizing Single-Purpose Processors

- Optimization: making design metric values the best possible
- Optimization opportunities

- Original program
- FSMD
- Datapath
- FSM

```
0: int x, y;  
1: while (1) {  
2:   while (!go_i);  
3:   x = x_i;  
4:   y = y_i;  
5:   while (x != y) {  
6:     if (x < y)  
7:       y = y - x;  
     else  
9:       x = x - y;  
   }  
9:   d_o = x;  
}
```



Optimization I: Optimizing the Original Program

- Analyze program attributes and look for areas of possible improvement
 - number of computations
 - size of variable
 - time and space complexity
 - operations used
 - multiplications and divisions are very expensive

original program

```
0: int x, y;  
1: while (1) {  
2:   while (!go_i);  
3:   x = x_i;  
4:   y = y_i;  
5:   while (x != y) {  
6:     if (x < y)  
7:       y = y - x;  
8:     else  
9:       x = x - y;  
10:  
11: }  
12: d_o = x;  
13: }
```

replace the subtraction operation(s)
with modulo operation in order to
speed up program

optimized program

```
0: int x, y, r;  
1: while (1) {  
2:   while (!go_i);  
3:   // x must be the larger number  
4:   if (x_i >= y_i) {  
5:     x=x_i;  
6:     y=y_i;  
7:   }  
8:   else {  
9:     x=y_i;  
10:    y=x_i;  
11:  }  
12:  while (y != 0) {  
13:    r = x % y;  
14:    x = y;  
15:    y = r;  
16:  }  
17:  d_o = x;  
18: }
```

GCD(42, 8) - 9 iterations to complete the loop

x and y values evaluated as follows : (42, 8), (43, 8),
(26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

GCD(42,8) - 3 iterations to complete the loop

x and y values evaluated as follows: (42, 8), (8,2), (2,0)

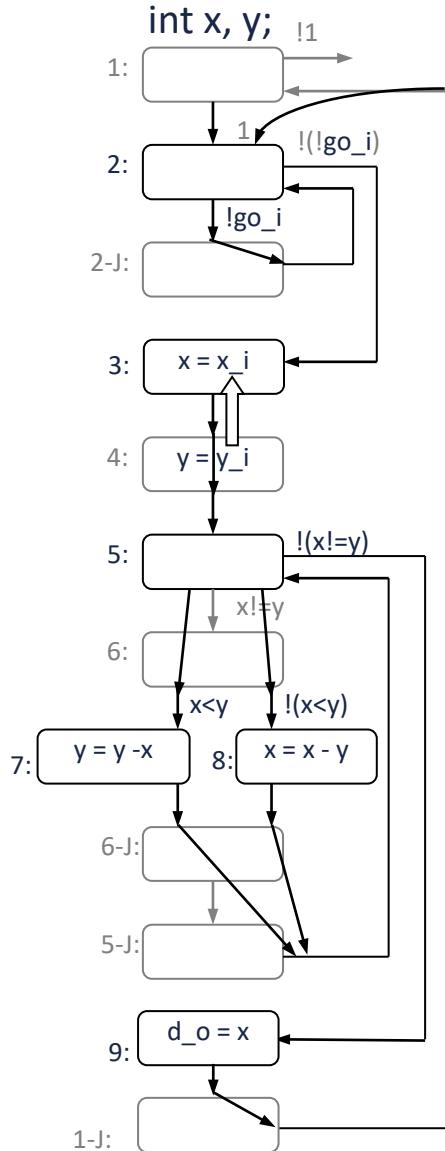
Optimization II: Optimizing the FSMD

Areas of possible improvements:

- Merge states
 - states with constants on transitions can be eliminated, transition taken is already known
 - states with independent operations can be merged
- Separate states
 - states which require complex operations ($a^*b^*c^*d$) can be broken into smaller states to reduce hardware size
- Scheduling

Optimization II: Optimizing the FSMD

Original FSMD



eliminate state 1 – transitions have constant values

merge state 2 and state 2J – no loop operation in between them

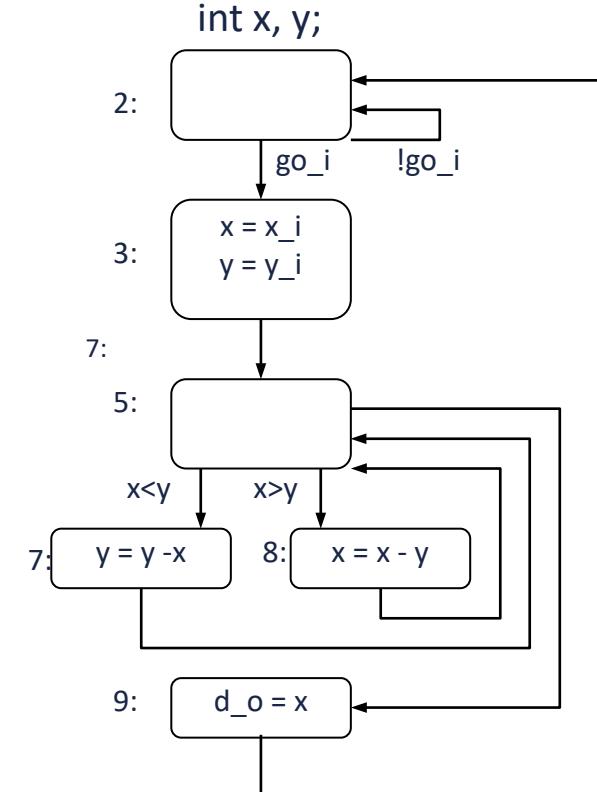
merge state 3 and state 4 – assignment operations are independent of one another

merge state 5 and state 6 – transitions from state 6 can be done in state 5

eliminate state 5J and 6J – transitions from each state can be done from state 7 and state 8, respectively

eliminate state 1-J – transition from state 1-J can be done directly from state 9

Optimized FSMD



Optimization III: Optimizing the Datapath

- Sharing of functional units
 - One-to-one mapping, as done previously, is not necessary
 - If same operation occurs in different states, they can share a single functional unit
- Multi-functional units
 - ALUs support a variety of operations, it can be shared among operations occurring in different states

Optimization IV: Optimizing the FSM

- State encoding
 - Task of assigning a unique bit pattern to each state in an FSM
 - Size of state register and combinational logic vary
 - Can be treated as an ordering problem
- State minimization
 - Task of merging equivalent states into a single state
 - State equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 3:

Language and Compiler Basics (I)

Sitao Huang

sitaoh@uci.edu

January 18, 2022

Slide courtesy of Prof. Vikram Adve, UIUC, CS 426: Compiler Construction



Tentative Schedule

- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): ***Language and Compiler Basics***
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compiler Optimizations for Accelerators
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

Languages and Compilers for Hardware Accelerators

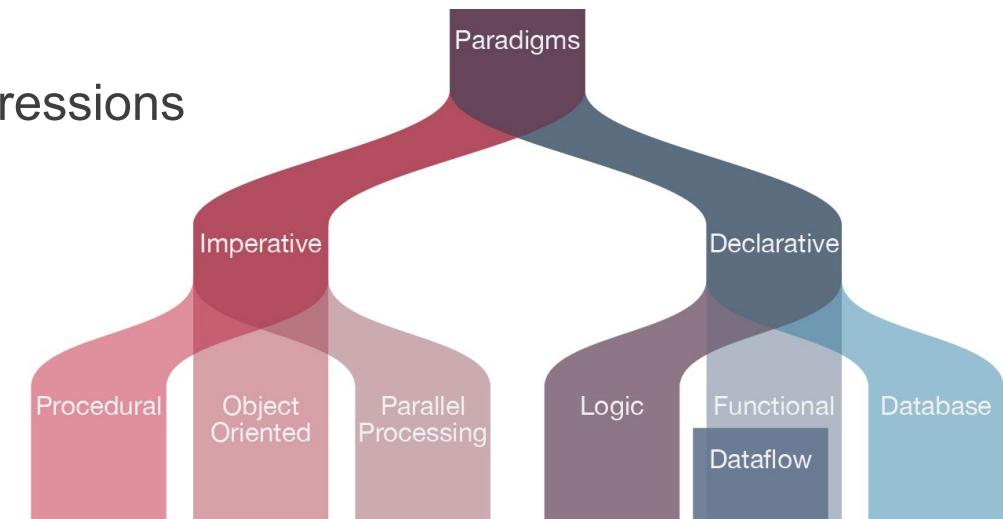
- **Programming languages:** formal language comprising a set of strings, used to implement algorithms
- Provide an abstraction for underlying hardware
- Provide a set of general operators to describe a range of applications
- Primitive elements of programming languages
 - Syntax: rules that define the correct combination of symbols

Lisp
example:
(from
Wikipedia)

```
expression ::= atom | list
atom      ::= number | symbol
number    ::= [+-]?['0'-'9']+
symbol    ::= ['A'-'Z'|'a'-'z'].*
list      ::= '(' expression* ')'
```

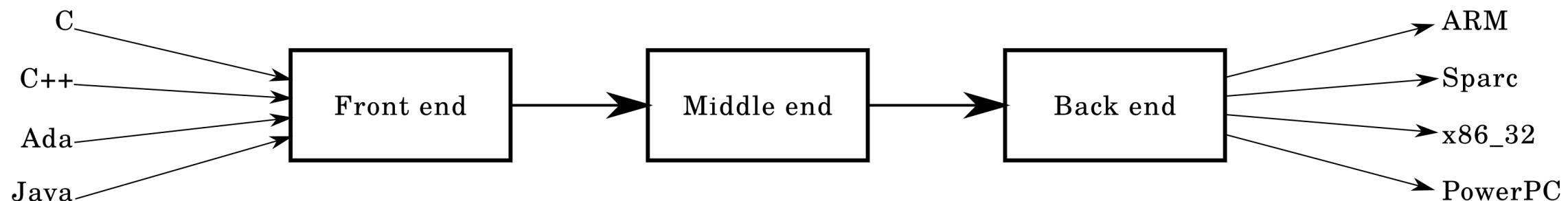
- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace);
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

- Semantics: meaning of languages
 - Type system: how a language classify values and expressions
 - Standard library and run-time system
- Categories
 - Typed vs untyped languages, static vs dynamic typing
 - Functional programming vs imperative programming
 -



Languages and *Compilers* for Hardware Accelerators

- **Compilers:** a special program that processes code in a particular programming language and translates input code into the code in another or the same language.
- Categories
 - Just-In-Time (JIT) compiler, Ahead-of-Time (AOT) compiler
 - Source-to-source compiler
 -
- Three-stage compiler structure
 - Front end: translate source code to intermediate representation (IR), manages symbol table
 - Middle end: compiler analysis (e.g., data-flow analysis) and optimization (e.g., loop transformation)
 - Back end: architecture specific optimizations, code generation



Compilers

- **Compilers:** a special program that processes code in a particular programming language and translates input code into the code in another or the same language.
- Examples:
 - C++ to x86 assembly
 - C++ to C
 - Java to JVM bytecode
 - C to C (or any language to itself)
 - Make code faster/smaller, instrumentation, etc.

Use of Compiler Technology

- **Code generation:** To translate a program in a high-level language to machine code for a particular processor
- **Optimization:** Improve program performance for a given target machine
- **Text formatters:** translate TeX to dvi, dvi to postscript, etc.
- **Interpreters:** “on-the-fly” translation of code, e.g., Java, Perl, csh, Postscript
- **Automatic parallelization or vectorization**
- **Debugging aids:** e.g., purify for debugging memory access errors
- **Performance instrumentation:** e.g., -pg option of cc or gcc for profiling
- **Security:** JavaVM uses compiler analysis to prove safety of Java code
- **Many more cool uses!** Hardware design / synthesis, power management, code compression, fast simulation of architectures, transparent fault-tolerance, . . .

Key: Ability to extract properties of a program (*analysis*),
and optionally transform it (*synthesis/transformation*)

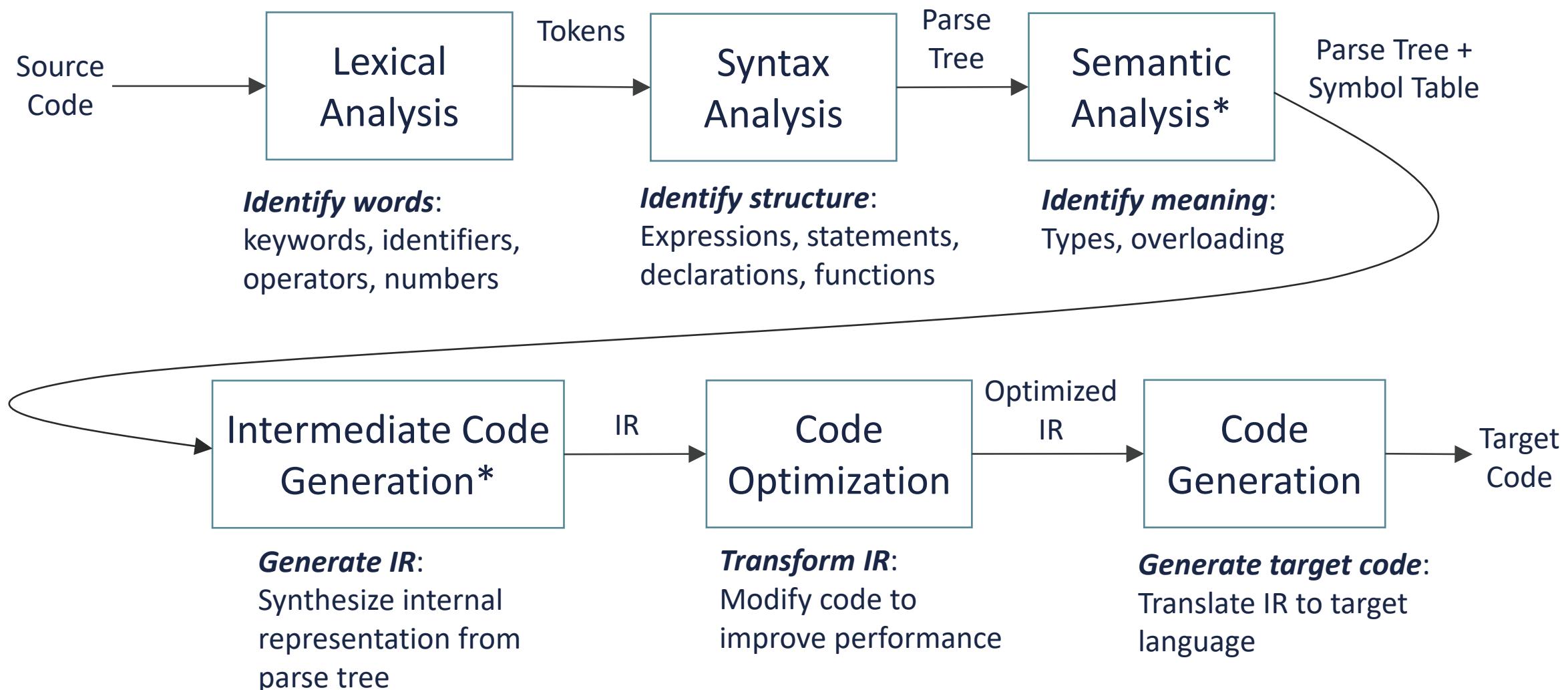
A Code Optimization Example

What machine-independent optimizations are applicable to the following C example? When are they safe?

```
1 int main() {  
2     ...  
3     X = ...;  
4     N = 1; i = 1;  
5     while (i <= 100) {  
6         j = i * 4;  
7         N = j * N;  
8         Y = X * 2.0;  
9         A[i] = X * 4.0;  
10        B[j] = Y * N;  
11        C[j] = N * Y * C[j];  
12        i = i + 1;  
13    }  
14    printArray(B, 400);  
15    printArray(C, 400);  
16 }
```

```
1 X = ...  
2 N = 1;  
3 j = 4;          // Induction Variable Substitution (SUBST),  
4           // Strength Reduction  
5 Y = X * 2.0;    // Loop-Invariant Code Motion (LICM)  
6 while (j <= 400) { // Linear Function Test Replacement (LFTR)  
7           // Dead Code Elimination (DCE) for i * 4  
8     N = j * N;  
9           // DCE of A, since A not aliased to B or C  
10    tmp = Y * N;  
11    B[j] = tmp;  
12    C[j] = tmp * C[j]; // Common Subexpression Elimination (CSE)  
13    j = j + 4;        // Induction Variable Substitution,  
14           // Strength Reduction  
15 }  
16 printArray(B, 400);  
17 printArray(C, 400);
```

General Structure of a Compiler

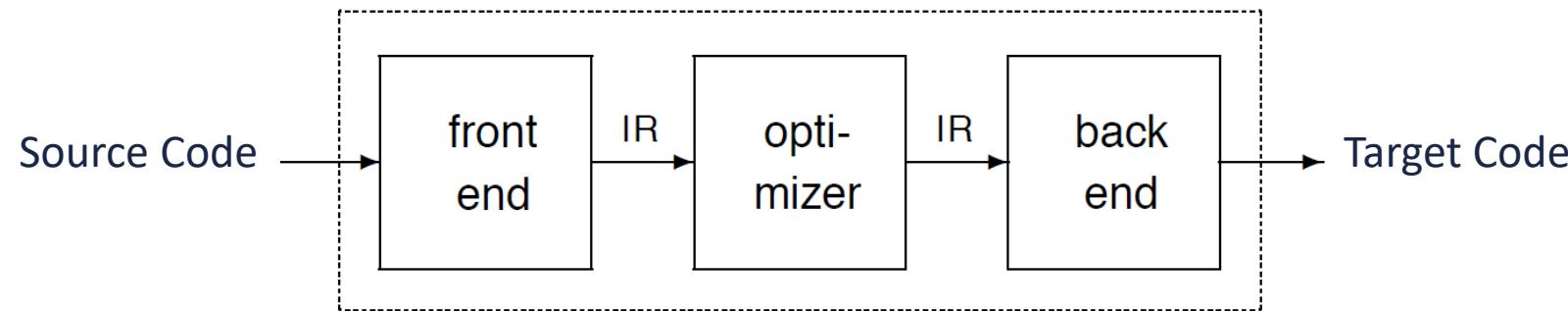


* Order varies

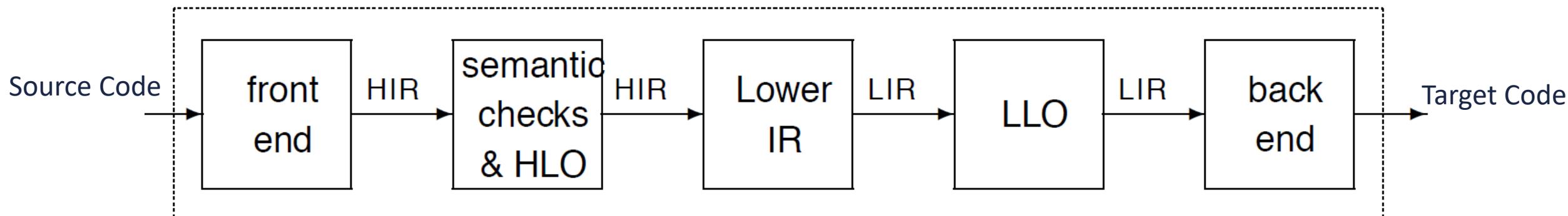
Intermediate Representation (IR)

- Intermediate Representation (IR) encodes all knowledge the compiler has derived about the source program.

Simple compiler structure



More typical compiler structure



Components and Design Goals for an IR

Components of IR

- *Code representation*: actual statements or instructions
- *Symbol table* with links to/from code
- *Analysis information* with mapping to/from code
- *Constants table*: strings, initializers, ...
- *Storage map*: stack frame layout, register assignments

Design Goals for an IR?

- No universally good IR
- The right choice depends strongly on the goals of the compiler

Common Code and Analysis Representations

- Code Representations
 - Usually have ***only one*** at a time
 - Common alternatives:
 - Abstract Syntax Tree (AST)
 - SSA form + CFG
 - 3-address code [+ CFG]
 - Stack code
 - Influences:
 - semantic information
 - types of optimizations
 - ease of transformations
 - speed of code generation
 - size
- Analysis Representations
 - May have ***several*** at a time
 - Common choices:
 - Control Flow Graph (CFG)
 - Symbolic expression DAGs
 - Data dependence graph (DDG)
 - SSA form
 - Points-to graph / Alias sets
 - Call graph
 - Influences:
 - analysis capabilities
 - optimization capabilities

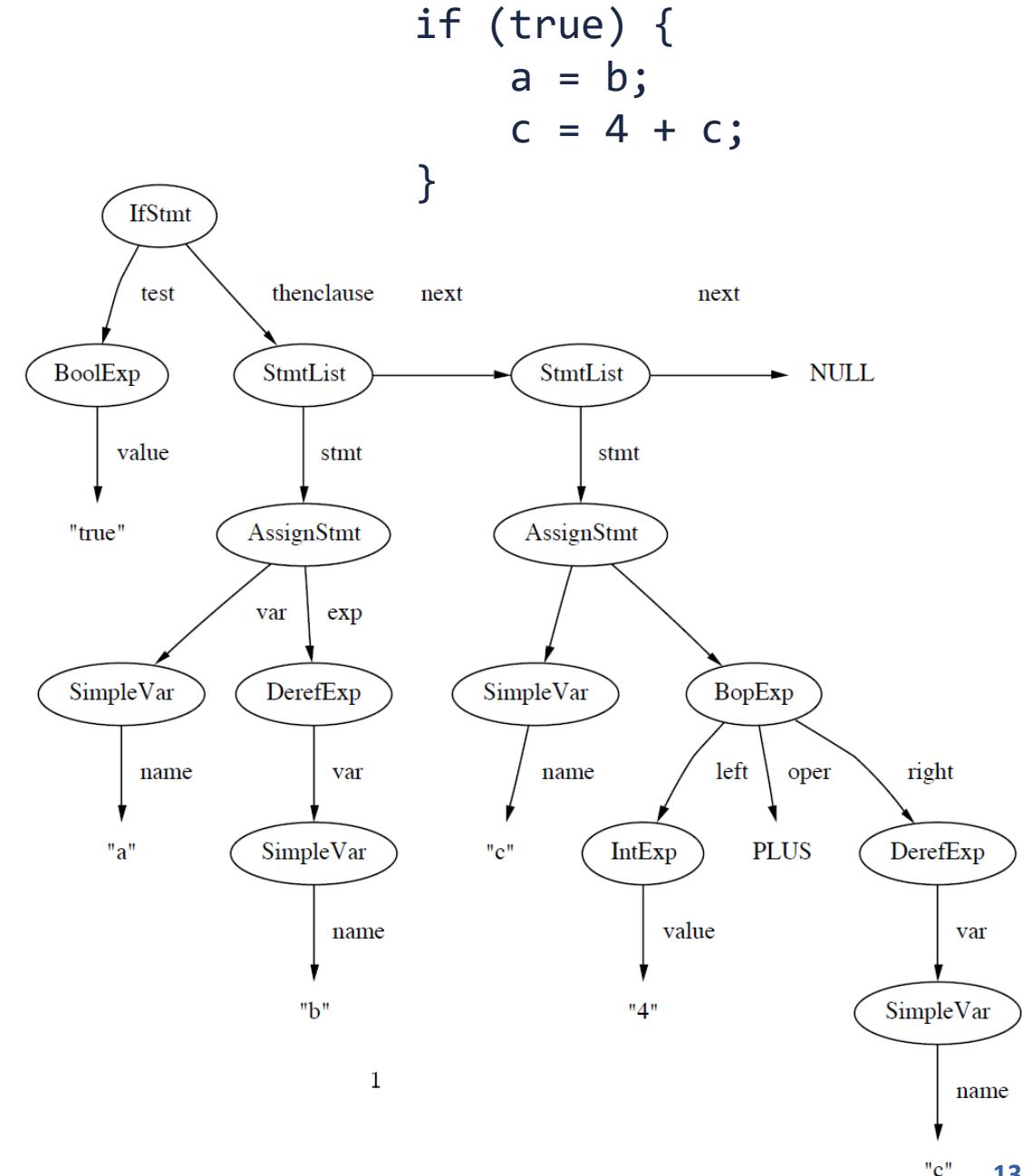
Categories of IRs By Structure

- Graphical IRs
 - Trees, directed graphs, DAGs
 - Node/edge data structures tend to be large
 - Harder to rearrange
 - Examples: AST, CFG, SSA, DDG, Expression DAG, Point-to graph
- Linear IRs
 - Pseudo-code for abstract machine
 - Many possible semantic levels
 - Simple, compact data structures
 - Easier to rearrange
 - Examples: 3-address, 2-address, accumulator, or stack code
- Hybrid IRs as the Code Representation
 - CFG + 3-address code (SSA or non-SSA)
 - AST (for control flow) + 3-address code (for basic blocks)
 - ...

C Instruction	2 address	3 address
$r = x;$	<code>mov r, x</code>	<code>mov r, x</code>
$r = x + y;$	<code>mov r, x</code> <code>add r, y</code>	<code>add r, x, y</code>

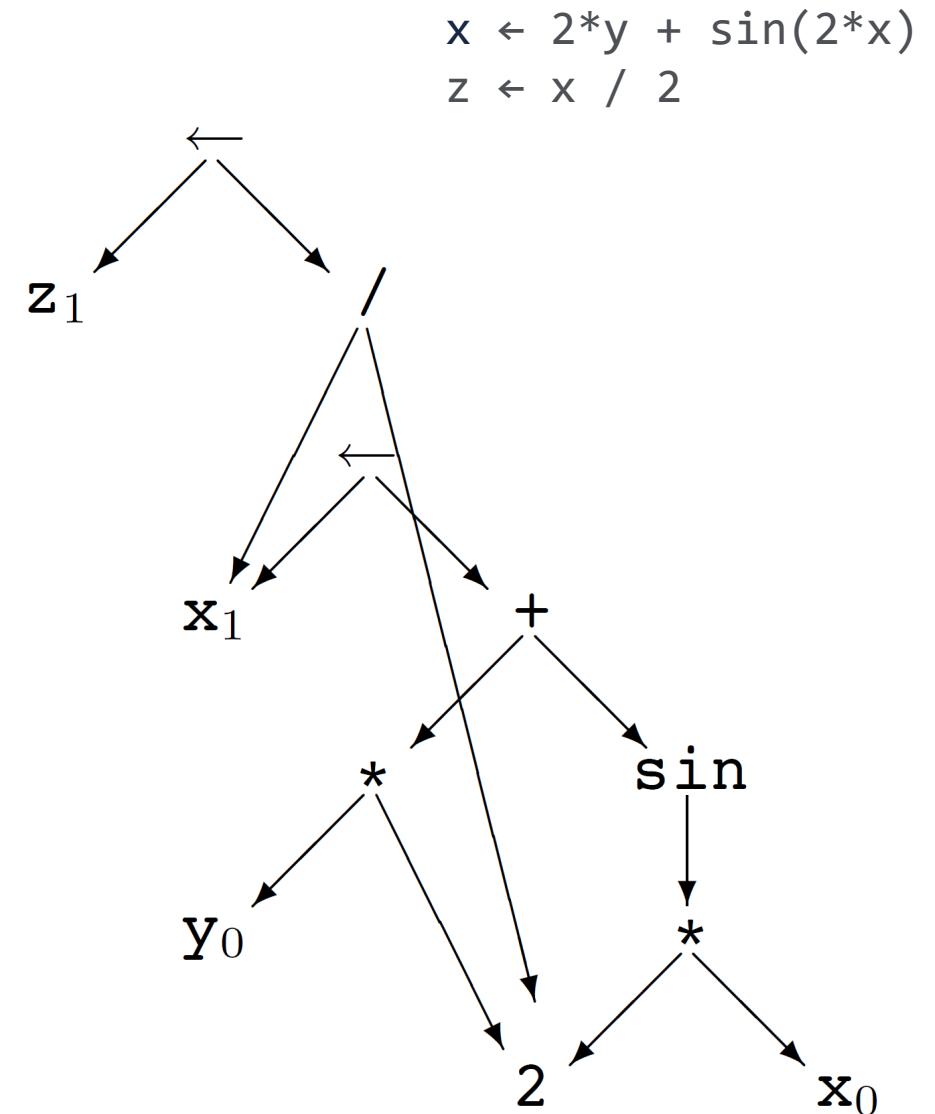
Abstract Syntax Tree (AST)

- Abstract Syntax Tree (AST): tree representation of the abstract syntactic structure of text (source code) written in a formal language
- It retains syntactic structure of the code
- Widely used in *source-source* compilers
- Captures both control flow constructs and straight-line code explicitly
- Traversal and transformations are both relatively expensive
 - Both are pointer-intensive
 - Transformation are memory-allocation-intensive



Directed Acyclic Graph (DAG)

- A Directed Acyclic Graph (DAG) is similar to an AST but with a unique node for each *value*.
- Advantages:
 - Sharing of values is explicit
 - Exposes redundancy (value computed twice)
 - Powerful representation for symbolic expressions
- Disadvantages:
 - Difficult to transform (e.g., delete a statement)
 - Not useful for showing control flow structure
 - Better for *analysis* than *transformation*



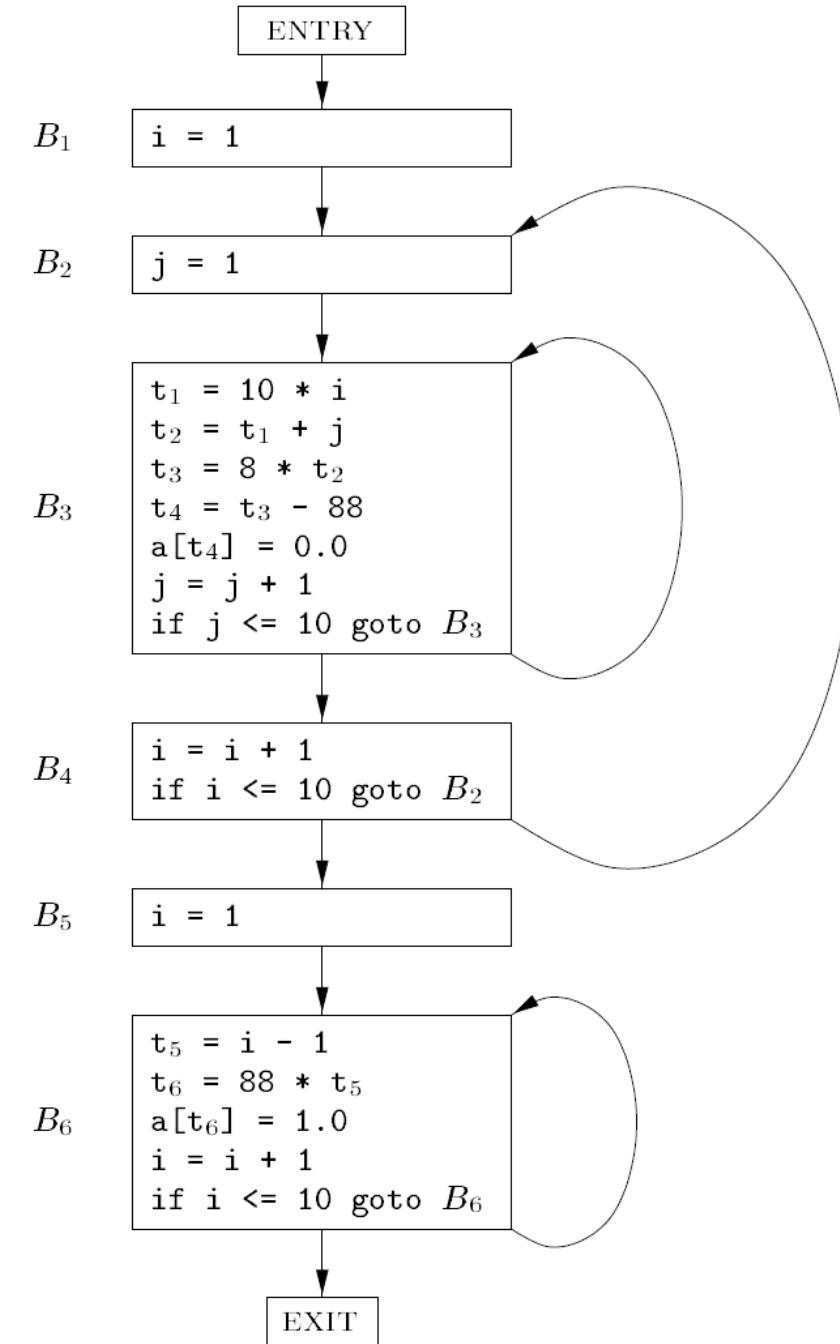
Control Flow Graph (CFG)

- **Basic Block:** a *maximal* consecutive sequence of statements (or instructions) $S_1 \dots S_n$ such that:
 - (a) the flow of control must enter the block at S_1 , and
 - (b) if S_1 is executed, then $S_2 \dots S_n$ are all executed in that order (unless one of the statements causes the program to halt)
 - **Leader:** the first statement of a basic block
 - **CFG:** a directed graph (usually for a single procedure) in which:
 - Each node is a single basic block
 - There is an edge $b_1 \rightarrow b_2$ if control **may** flow from last statement of b_1 to first statement of b_2 in some execution
- Note:** A CFG is a conservative approximation of the control flow!

Control Flow Graph (CFG)

- Example:

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Dominance in Control Flow Graphs

- **Dominates:** B_1 dominates B_2 iff all paths from entry node to B_2 include B_1
- Intuitively, B_1 is always executed before executing B_2 (or $B_1 = B_2$)

Which assignments dominate $(X+Y)$?

```
X = 1;  
if (...) {  
    Y = 4;  
}  
... = X + Y;
```

Which assignments dominate $(X+Y)$?

```
X = 1;  
if (...) {  
    Y = 4;  
    ... = X + Y;  
}
```

Static Single Assignment (SSA) Form

- Static Single Assignment (SSA) is a property of an IR: each variable is assigned exactly once, and every variable is defined before it is used.
- Informally, a program can be converted into SSA form as follows:
 - Each assignment to a variable is given a unique name
 - All of the uses reached by that assignment are renamed
- Easy for straight-line code:

$$\begin{aligned}V &\leftarrow 4 \\&\quad \leftarrow V + 5 \\V &\leftarrow 6 \\&\quad \leftarrow V + 7\end{aligned}$$
$$\begin{aligned}V_0 &\leftarrow 4 \\&\quad \leftarrow V_0 + 5 \\V_1 &\leftarrow 6 \\&\quad \leftarrow V_1 + 7\end{aligned}$$

- What about flow of control?
 - Introduce ϕ -functions

SSA with Control Flow

Two-way branch

```
if (...)           if (...)  
    X = 5;         X0 = 5;  
else             else  
    X = 3;         X1 = 3;  
Y = X;           X2 =  $\phi(X_0, X_1)$ ;  
                  Y0 = X2;
```

While loop

```
j = 1;  
S: // while (j < x)  
    if (j >= X)  
        goto E;  
    j = j+1;  
    goto S  
E:  
    N = j;
```

```
j5 = 1;  
j2 =  $\phi(j_5, j_4)$ ;  
if (j2 >= X)  
    goto E;  
j4 = j2+1;  
goto S  
E:  
N = j2;
```

SSA with Control Flow

- ϕ -functions: In a basic block B with N predecessors, P_1, P_2, \dots, P_N ,
$$X = \phi(V_1, V_2, \dots, V_N)$$

assigns $X = V_j$ if control enters block B from P_j , $1 \leq j \leq N$.

- Properties of ϕ -functions:
 - ϕ is not an executable operation
 - ϕ has exactly as many arguments as the number of incoming BB edges
 - Think about ϕ argument V_i as being evaluated on CFG edge from predecessor P_i to B
- **SSA form definition:**

A program is in SSA form if:

- Each variable is assigned a value in exactly one statement
- Each use of a variable is *dominated* by the definition

Tradeoffs of SSA Form

Strengths:

- Each use is reached by a single definition (simpler analyses)
- Def-use pairs are explicit: compact dataflow information
- No *write-after-read* and *write-after-write* dependences
- Can be directly transformed during optimizations

Many dataflow
optimizations are
much faster

Weaknesses:

- Space requirement: many variables, many ϕ Functions
- Limited to scalar values; an array is treated as one big scalar
- When target is low-level machine code, limited to “virtual registers” (memory is not in SSA form)
- Copies introduced when converting back to real code

Stack Machine Code

- Used in compilers for stack architectures
- Popular again for bytecode languages, e.g., in JVM
- Advantages:
 - Compact form
 - Introduced names are implicit, not explicit
 - Simple to generate and execute code
- Disadvantages:
 - Does not match current architectures
 - Many spurious dependences due to stack
 - Difficult to reordering transformations
 - Cannot “reuse” expressions easily (must store and re-load)
 - Difficult to express optimized code

Example

$$x - 2 * y - 2 * z$$

Stack machine code:

```
push x
push 2
push y
multiply
push 2
push z
multiply
add
subtract
```

Three Address Code

- Three Address Code: a term used to describe many different representations where each statement is single operator and there are at most three operands
- Advantages:
 - Compact and very uniform
 - Makes intermediates values explicit
 - Suitable for many levels (high, mid, low)
- Disadvantages:
 - Large name space (due to temporaries)
 - Loses syntactic structure of source

Example

```
if (x > y)
    z = x - 2 * y
```

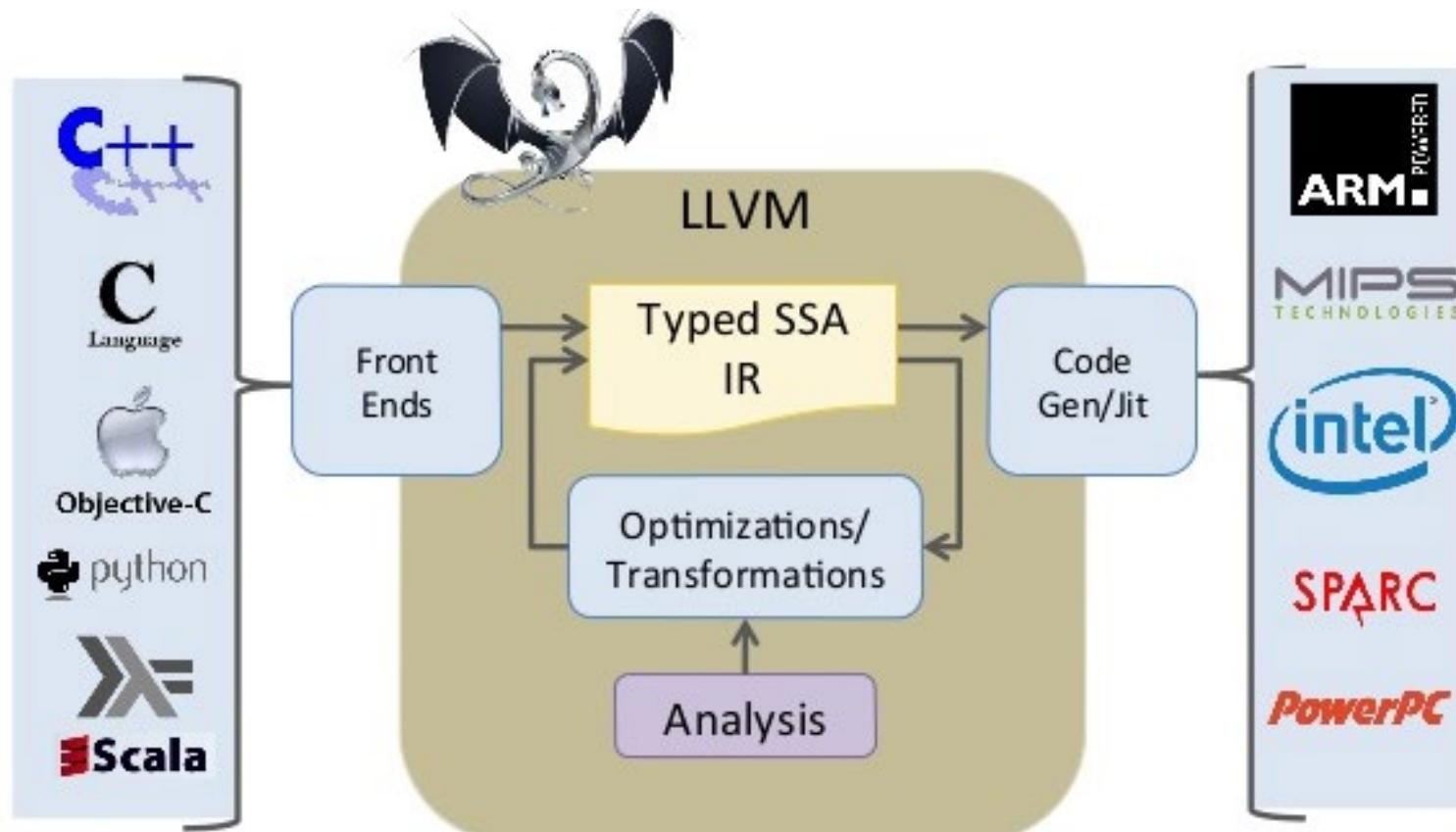
3-address code:

```
t1 ← load x
t2 ← load y
t3 ← t1 gt t2
br t3 L2 L1
L1: t4 ← 2 * t2
      t5 ← t1 - t4
      z ← store t5
L2: ...
```

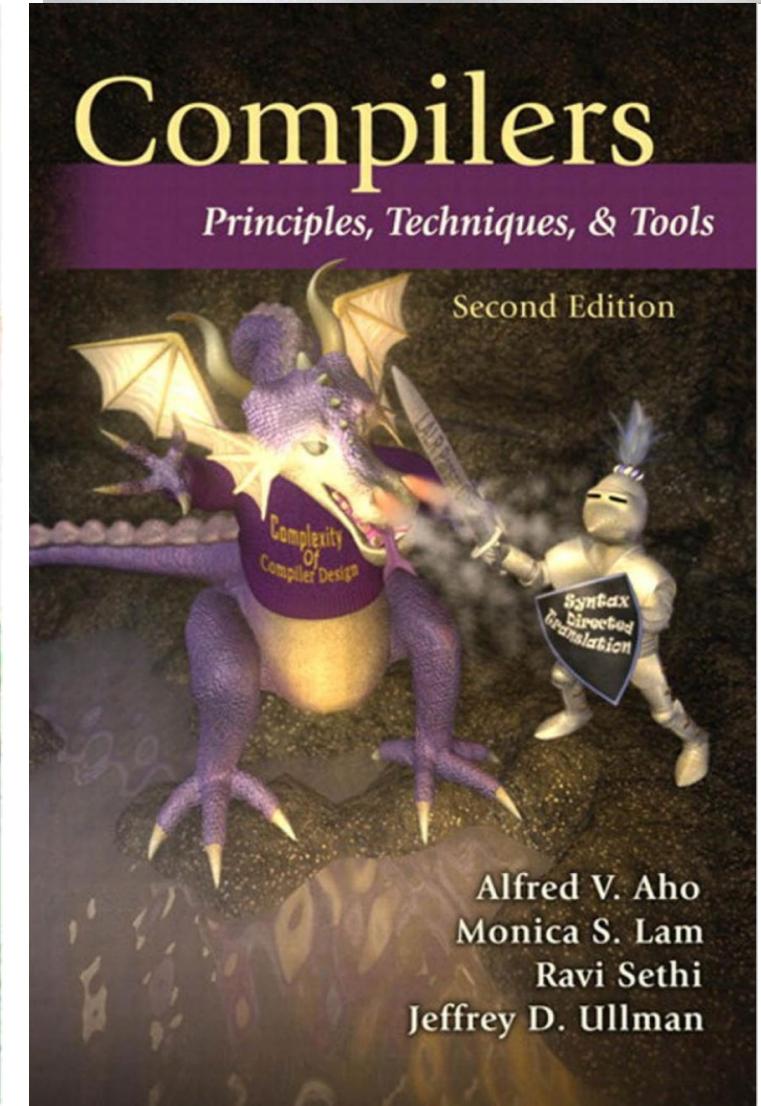
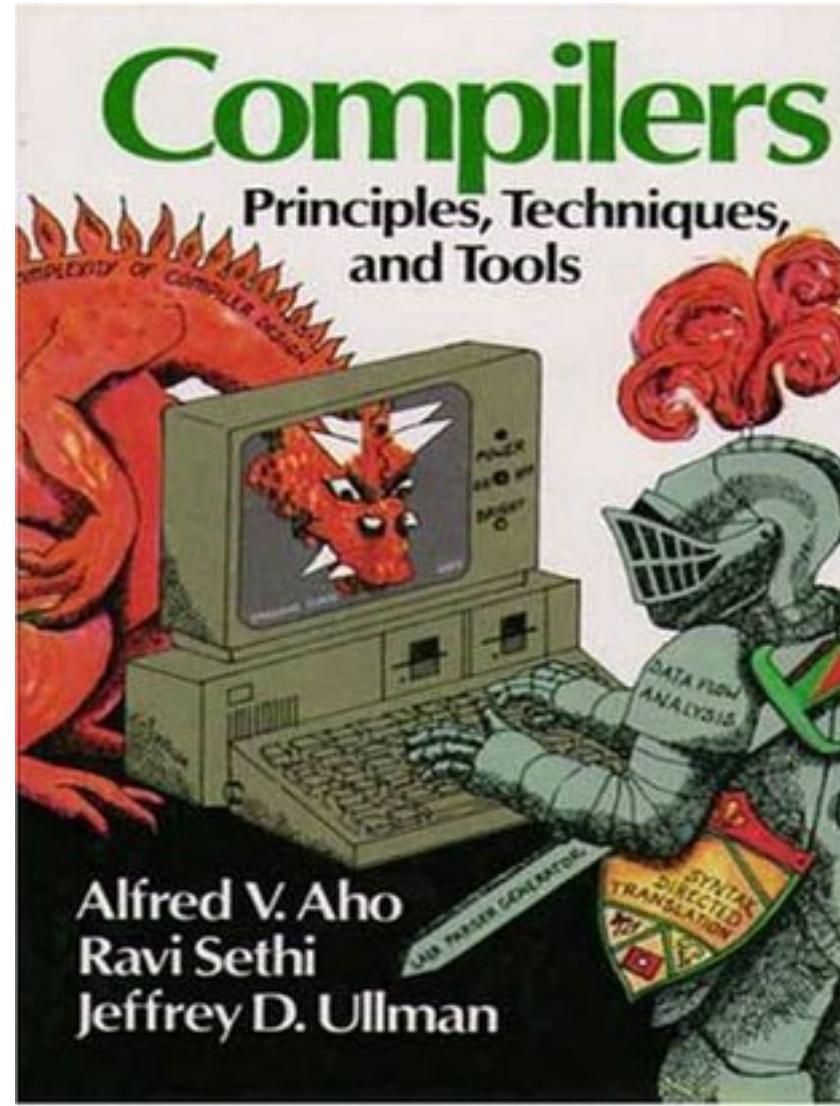
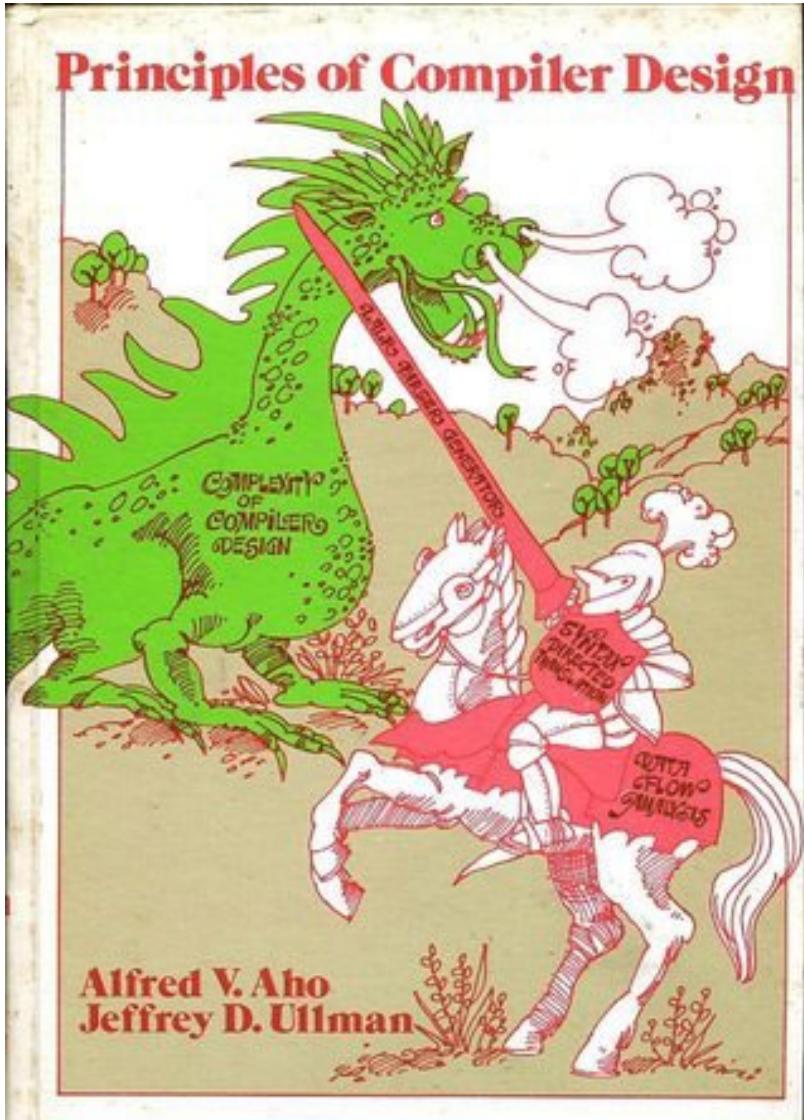
The LLVM Compiler Infrastructure



- *The LLVM Project*: a collection of modular and reusable compiler and toolchain technologies
- LLVM: the name is not an acronym; originally represents “Low Level Virtual Machine”
- Started in 2000 at the *University of Illinois at Urbana-Champaign*, under the direction of *Vikram Adve* and *Chris Lattner*.



Compilers



EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 3:

Language and Compiler Basics (II)

Sitao Huang

sitaoh@uci.edu

January 25, 2022

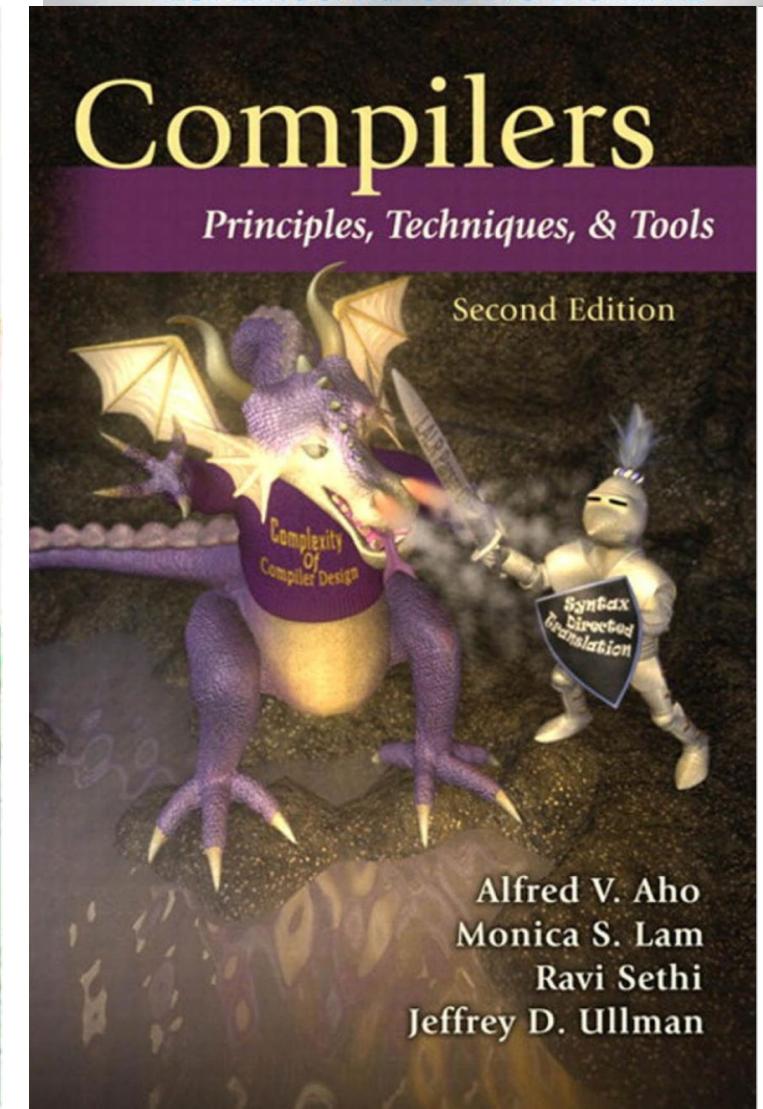
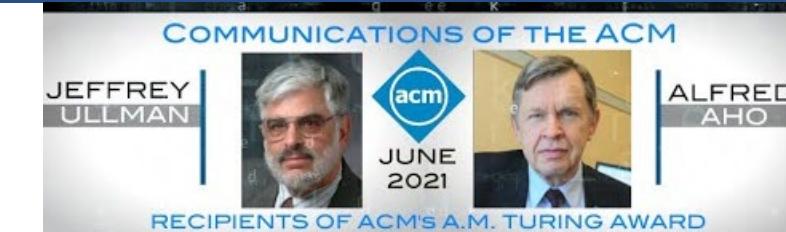
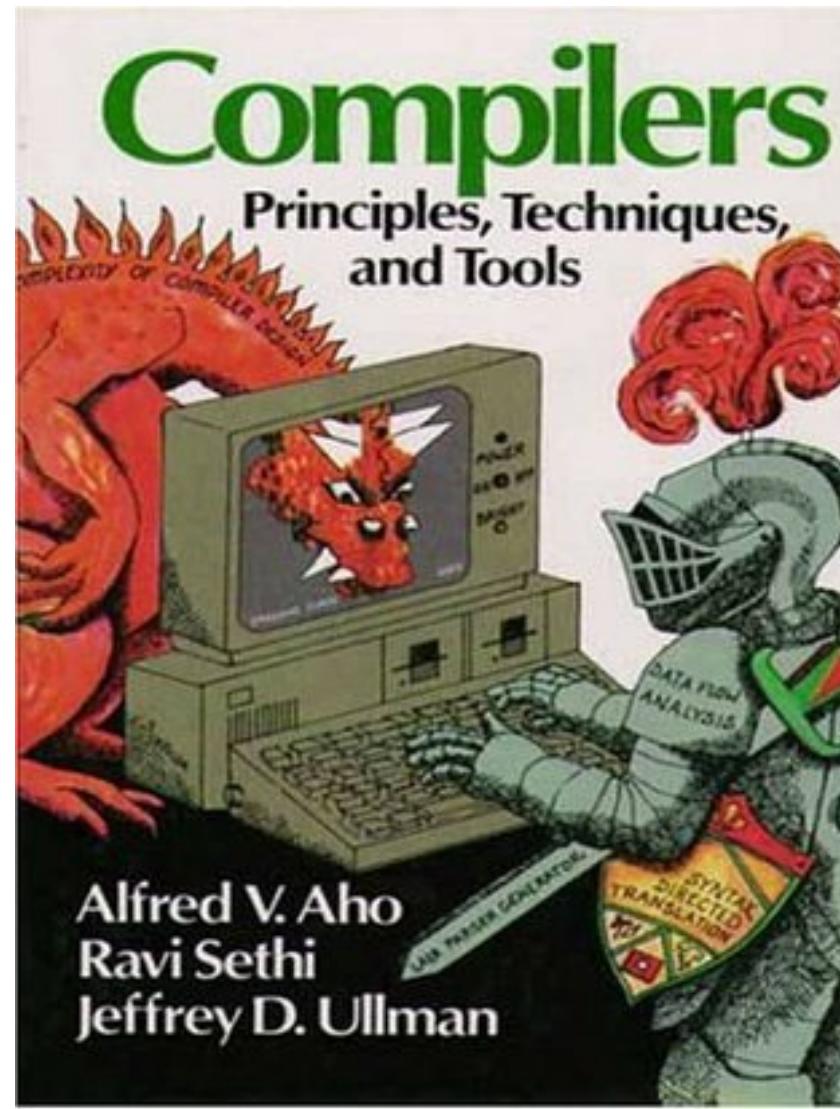
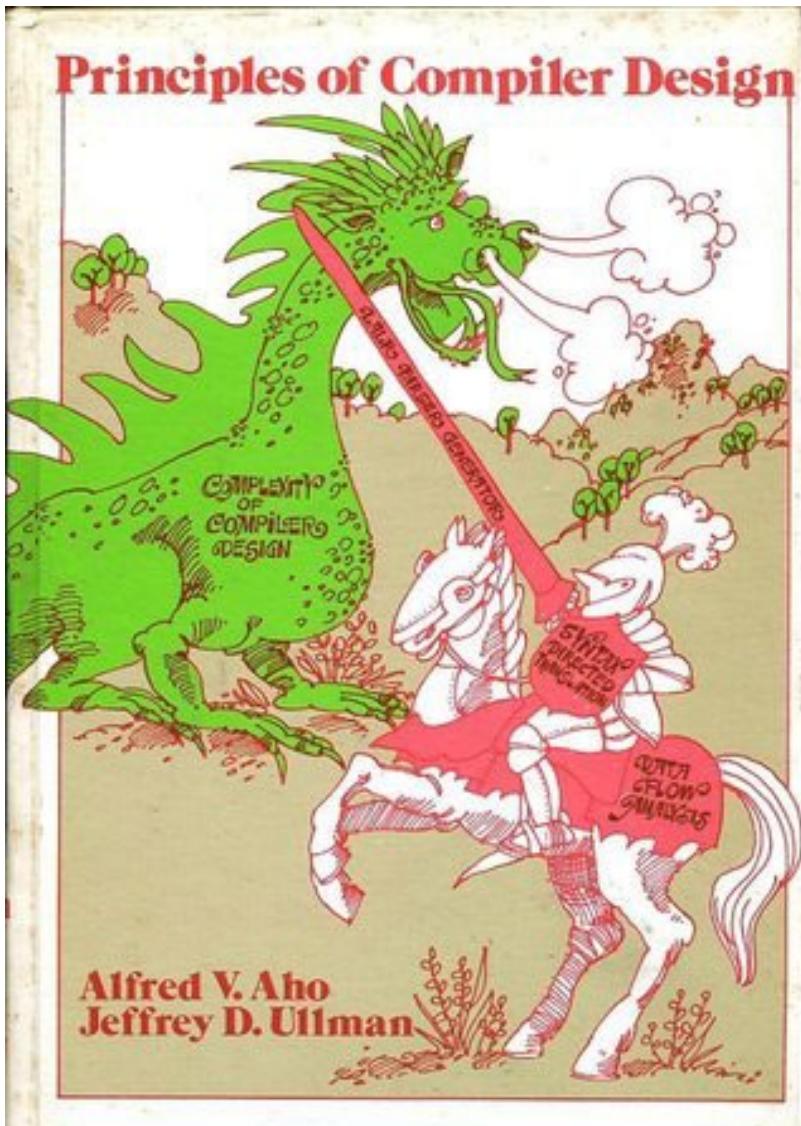
Slide courtesy of Prof. Vikram Adve, UIUC, CS 426: Compiler Construction



Logistics

- **Homework 1** released, due: ***January 31***, 11:59 PM on Canvas
- Homework 2 expected to release later this week, due February 7
- **Midterm: February 10** (Thursday), 8:00-9:20 AM (in class)
- Midterm review session / Q&A: February 8 (Tuesday)
- Project proposal due: February 14
 - Options: (a) literature review paper or (b) compiler + accelerator project
- In-person instruction starting next week (week 5)!
- First in-person class: ***February 1***

Compilers



Flow Graphs

- **Flow Graph:** A triple $G = (N, A, s)$ where (N, A) is a (finite) directed graph, $s \in N$ is a designated entry node, and there is a path from node s to every node $n \in N$ (s dominates every node $n \in N$).
- **Entry node:** A node with no predecessors
- **Exit node:** A node with no successors

Properties

- There is a unique entry node, which must be s (reachability assumption)
- Conservative: some branches may never be taken
- Control flow graphs are usually sparse. That is, $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.

Dominance in Flow Graphs

Definitions

- d dominates n (“ $d \text{ dom } n$ ”) iff every path in G from s to n contains d .
- d properly dominates n if d dominates n and $d \neq n$.
- d is the immediate dominator of n (“ $d \text{ idom } n$ ”) if d is the last dominator on any path from initial node to n , $d \neq n$.

Properties

- $s \text{ dom } d$, \forall node d in G .
- Partial Ordering: The dominance relation of a flow graph G is a partial ordering:
 - Reflective: $n \text{ dom } n$ is true $\forall n$.
 - Antisymmetric: if $d \text{ dom } n$, then $n \text{ dom } d$ cannot hold.
 - Transitive: $(d_1 \text{ dom } d_2) \wedge (d_2 \text{ dom } d_3) \Rightarrow d_1 \text{ dom } d_3$

Dominator Tree

- The dominators of a node form a chain:
- If $d_1 \text{ dom } n$ and $d_2 \text{ dom } n$, and $d_1 \neq d_2$, then $d_1 \text{ dom } d_2$ **or** $d_2 \text{ dom } d_1$.
 \Rightarrow Every node $n \neq s$ has a **unique** immediate dominator.

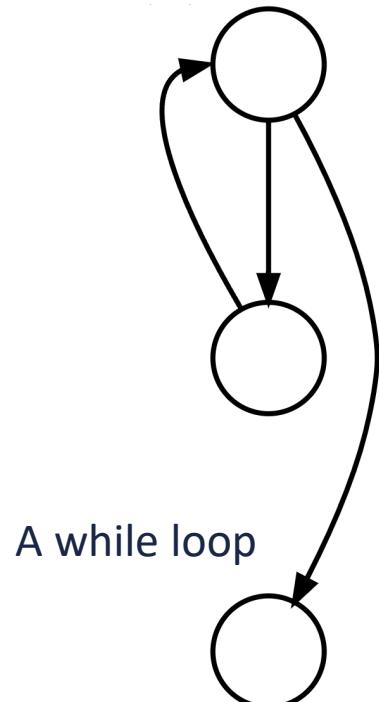
Dominator Tree

The *Dominator Tree* of a flow graph G is a graph with the same nodes as G , and an edge $n_1 \rightarrow n_2$ iff $n_1 \text{ dom } n_2$.

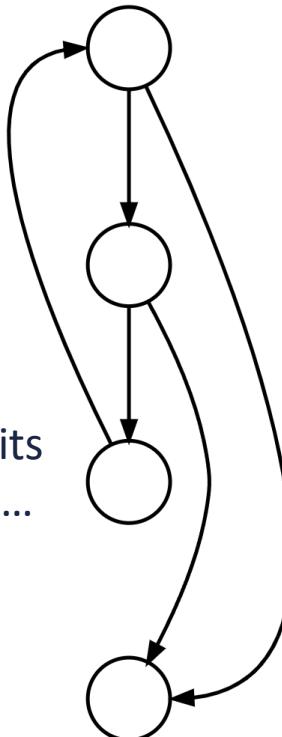
Defining Loops in Flow Graphs

Why defining loops is challenging?

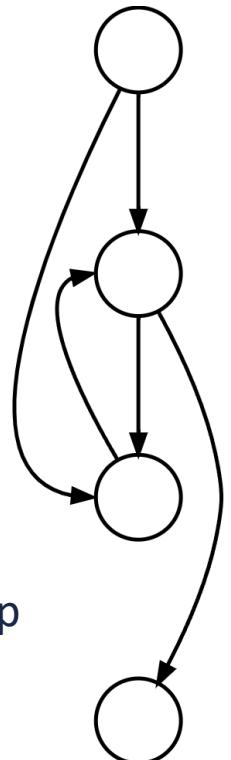
- Easy case: structured nested loops: FOR loop or WHILE loop
- Harder case: arbitrary flow and exits in loop body, but unique loop “entry”
- Hardest case: no unique loop “entry” (“irreducible loops”)



A natural loop with two exits
(e.g., while loop with an if ...
break in the middle), non-
structured but reducible

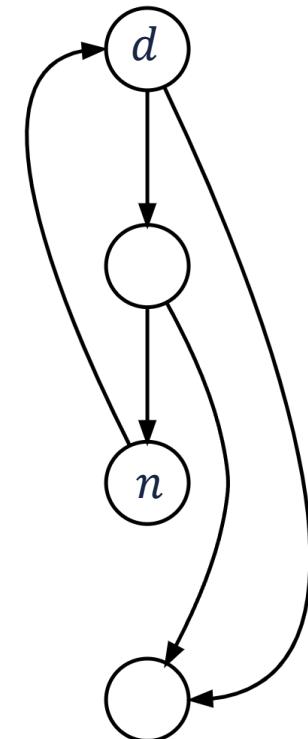


An irreducible CFG, a loop
with two entry points, e.g.,
goto into a while or for loop



Defining Loops in Flow Graphs

- **Back Edge:** An edge $n \rightarrow d$ where $d \text{ dom } n$
- **Natural Loop:** Given a back edge, $n \rightarrow d$, the *natural loop* corresponding to $n \rightarrow d$ is the set of nodes: $\{d\} \cup \{n\} \cup \{x \mid x \text{ can reach } n \text{ without going through } d\}$
- **Loop Header:** A node d that dominates all nodes in the loop
 - Header is unique for each natural loop
 $\Rightarrow d$ is the unique entry point into the loop
 - Uniqueness is very useful for many optimizations



Preheader: An Optimization Convenience

The Idea:

- If a loop has multiple incoming edges to the header, moving code out of the loop safely is complicated
- Preheader gives a safe place to move code before a loop

The Transformation:

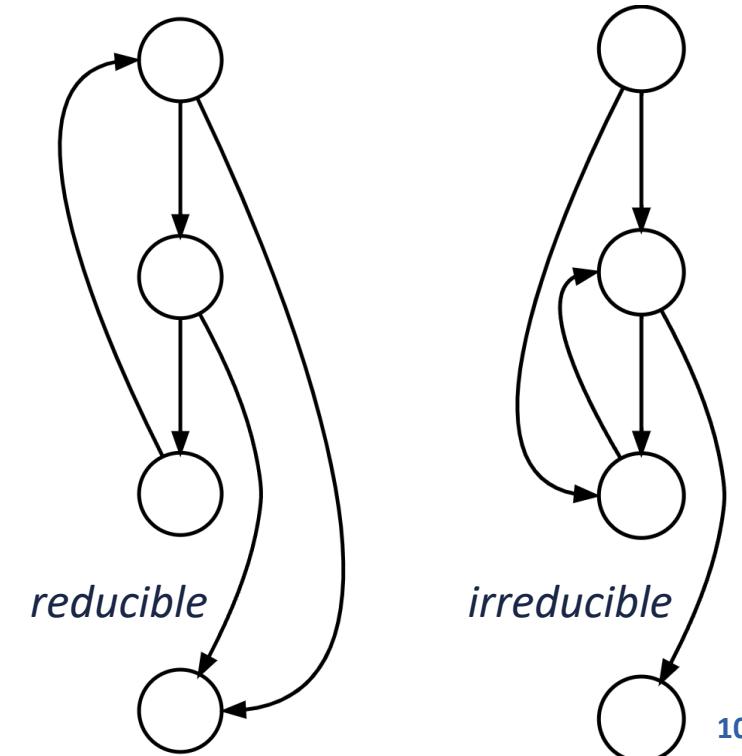
- Introduce a pre-header p for each loop (let loop header be d):
 - Insert node p with one out edge: $p \rightarrow d$
 - All edges that previously entered d should now enter p

Reducible and Irreducible Flow Graphs

Reducible Flow Graph:

A flow graph G is called reducible iff we can partition the edges into two sets:

- *Forward edges*: should form a DAG in which every node is reachable from initial node
- Other edges must be *back edges*: i.e., only these edges $n \rightarrow d$ where $d \text{ dom } n$
- Otherwise, graph is called *irreducible*
- *Idea*:
- Every “cycle” has at least one back edge
 \Rightarrow All “cycles” in a reducible graph are natural loops
 (NOT true in an irreducible graph!)



Dataflow Analysis

- A technique for collecting information about the flow of values and other program properties over control-flow paths
- Examples
 - What definitions of x reach a given use of x (and vice versa)?
 - Are any uses research by a particular definition of x ?
 - What $\langle \text{ptr}, \text{target} \rangle$ pairs are possible at each statement?
 - Is variable x defined on every path to a use of x ?
 - Is a pointer to a local variable live on exit from a procedure?
- Applications
 - Pointer Analysis
 - Type inference
 - Common Subexpression Elimination (CSE)
 - Loop-invariant code motion
 - ...

Definitions

- ***Alias or alias pair***: two different names for the same storage location
- ***Reference***: An occurrence of a name in a program statement
- ***Use of a variable***: A reference that *may* read the value of the variable
- ***Definition of a variable***: A reference that *may* store a value into the storage
 - Unambiguous definition: guaranteed to store to X
 - Ambiguous definition: may store to X

Ambiguity comes from aliases, unpredictable side effects of procedure calls, arrays

Dataflow Analysis Basics

- **Point:** A location in a basic block just before or after some statement
- **Path:** A path from p_1 to p_n is a sequence of points p_1, p_2, \dots, p_n such that (intuitively) some execution can visit these points in order
- **Kill of a definition:** A definition d of variable V is killed on a path if there is an unambiguous definition of V on that path
- **Kill of an expression:** An expression e is killed on a path if there is a possible definition of any of the variables of e on that path

An Example Dataflow Problem: Reaching Definitions

- **Reaching Definitions**
 - $\forall p$, Compute $\text{REACH}(p)$: the set of defs that reach point p .
 - Definition d reaches points p if there is a path from the point after d to p such that d is not killed along that path
 - **Dataflow Variables** (for each basic block B)
 - $\text{Gen}(B)$: the set of defs in B that are not killed in B
 - $\text{Kill}(B)$: the set of all defs that are killed in B
 - $\text{In}(B)$: the set of defs that reach the point before first statement in B
 - $\text{Out}(B)$: the set of defs that reach the point after last statement in B
- 
- }
- Local properties of B
- }
- Global dataflow
properties

Dataflow Analysis for Reaching Definitions

- Dataflow Equations

$$In(B) = \bigcup_{p:p \rightarrow B} Out[p]$$

Confluence operator

$$Out[B] = Gen[B] \bigcup (In[B] - Kill[B])$$

- Dataflow Algorithms

Goal: solve these $2n$ simultaneous dataflow equations (n is # of basic blocks)

- Block-structured graph (no GOTO; no BREAK from loops):
 - bottom-up evaluation, one scope at a time
- General flow graphs:
 - Iterative solution

Iterative Algorithm for Reaching Definitions

1. Initialize:

```
/* If there are globals or formals, in[s] ≠ ϕ */  
in[B] = ϕ      ∀B  
out[B] = gen[B] ∀B
```

2. Iterate until Out[B] does not change:

```
do  
    change = false  
    for each block B do  
         $In(B) = \bigcup_{p:p \rightarrow B} Out[p]$   
        oldout =  $Out[B]$   
         $Out[B] = Gen[B] \cup (In(B) - Kill[B])$   
        if ( $oldout \neq Out[B]$ ) change = true  
    end  
while (change == true)
```

Convergence of the Algorithm

$\text{Out}[B]$ converges in a finite number of iterations. Why?

- $\text{Out}[B]$ is finite $\forall B$
- $\text{Out}[B]$ never decrease $\forall B$
 - Only KILL sets (constants) are ever subtracted from OUT sets
 - IN sets never decrease (if OUT sets never decrease)

Acyclic Property

- Definitions need propagate only over acyclic paths
 - Each block only adds $\text{Gen}[B]$, subtracts $\text{Kill}[B]$
 - $\cup, -$: only need to add, remove once
 - Must visit each block exactly once
- Also need one final iteration to check convergence

Assume reducible graphs \rightarrow cycles formed by back edges

- No back edges: 2 iterations
- 1 back edge (on any acyclic path): 3 iterations
- K back edges on any acyclic path: $k + 2$ iterations

Another Example: Available Expressions

- Available Expressions: $x + y$ is available at point p if:
 - Every path to p evaluates $x + y$
 - Between the last such evaluation and p on each path, neither x nor y is modified

Kill: Block B kills $x + y$ if it may assign to x or y , and it does not subsequently recompute $x + y$

Generate: Block B generates $x + y$ if it definitely evaluates $x + y$, and it does not subsequently modify x or y

Dataflow variables:

Let U = universal set of expressions in the program. Then:

$$in[B] = \{\epsilon \in U | \epsilon \text{ is available at entry to } B\}$$

$$out[B] = \{\epsilon \in U | \epsilon \text{ is available at exit to } B\}$$

$$e_kill[B] = \{\epsilon \in U | \epsilon \text{ is generated by } B\}$$

$$e_kill[B] = \{\epsilon \in U | \epsilon \text{ is killed by } B\}$$

Naming Expressions

Examples

```
1 a = x * y;                                // eval e_1: x * y
2 b = x * y;                                // eval e_1: x * y: redundant
3 x = 2;                                     // "kills" e_1
4 c = x * y;                                // eval e_1: x * y
5
6 if (...) { x=5; t= x+y; }                  // eval e_2: x+y
7 else      { x=9; t= x+y; }                  // eval e_2: x+y
8 x = x+y;                                  // eval e_2: x+y: redundant!
9
10 p = cond? &x : &z;
11 ... = *p + 1;                            // e_3: X+1, e_4: Y+1 may not be eval
12 ... = X + 1;                            // eval e_3: X+1 may not be redundant
```

Dataflow Analysis for Available Expressions

- Dataflow Equations:

$$In(B) = \bigcap_{p:p \rightarrow B} Out[p]$$

$$Out[B] = e_gen[B] \bigcap (In[B] - e_kill[B])$$

- Iterative Algorithm: algorithm is identical to *Reaching Definitions* except:
 - *Confluence* operator is \cap instead of \cup
 - Algorithm must initialize sets as follows:

$$In[s] = \emptyset$$

$$Out[s] = e_gen[s]$$

$$Out[B] = U - e_kill[B], \forall B \neq s$$

General Approach to Dataflow Analysis

- Step 1: Choose dataflow variables for problems of interest:
 - $\text{Gen}[B]$: “information” generated in block B
 - $\text{Kill}[B]$: “information” killed in block B
 - $\text{In}[B], \text{Out}[B]$
- Step 2: Set up dataflow equations
 - What is the transfer function for each block?
e.g., $\text{Out}[B] = \text{Gen}[B] \cup (\text{In}[B] - \text{Kill}[B])$
 - Is it a forward or backward problem?
e.g., $\text{In}(B) = \bigcup_{p:p \rightarrow B} \text{Out}[p]$ or $\text{In}(B) = \bigcap_{p:p \rightarrow B} \text{Out}[p]$
 - What is the “confluence” operator?
e.g., \cup, \cap , or other
- Step 3: Solve iteratively until convergence
 - Postorder (PO) or Reverse Postorder (RPO)

Def-Use and Use-Def Chains

- Use-Def chain or ud-chain: For each use u of a variable v , $\text{DEFS}(u)$ is the set of instructions that may have defined v last prior to u
- Def-Use chain or du-chain: For each use d of a variable v , $\text{USES}(d)$ is the set of instructions that may use the value of v computed at d

Note: $d \in \text{DEFS}(u)$ iff $u \in \text{USES}(d)$

Note: du-chains (or ud-chains) form a graph

Comparing with SSA:

- Multiple defs reach each use, unlike SSA
- More edges in def-use graph than in SSA graph
- Fewer variable names, no ϕ functions

Constructing and Using du-Chains and ud-Chains

Construction:

- Construct DEFS(u) from the results of Reaching Definitions
- Invert DEFS to compute USES
 - ⇒ we can build chains very efficiently!

Some applications of chains:

- Building live ranges for graph-coloring register allocation
- Constant propagation
- Dead-code elimination
- Loop-invariant code motion

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 4:

Reconfigurable Accelerators

Sitao Huang

sitaoh@uci.edu

January 27, 2022

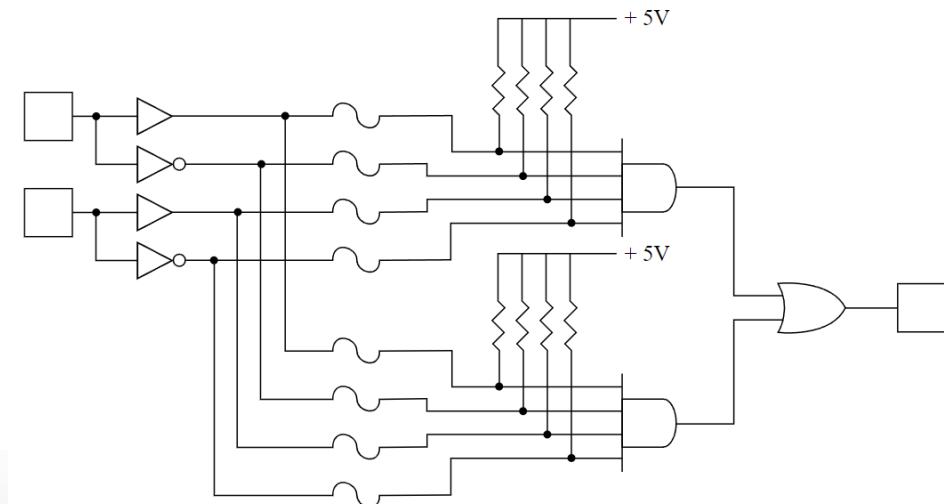


Logistics

- Homework 1 released, due: ***January 31***, 11:59 PM on Canvas
- Homework 2 expected to release later this week, due February 7
- **Midterm: February 10** (Thursday), 8:00-9:20 AM (in class)
- Midterm review session / Q&A: February 8 (Tuesday)
- Project proposal due: February 14
 - Options: (a) literature review paper or (b) compiler + accelerator project
- In-person (***hybrid***) instruction starting next week (week 5)!
- First in-person class: ***February 1***

Programmable Logic Devices (PLD)

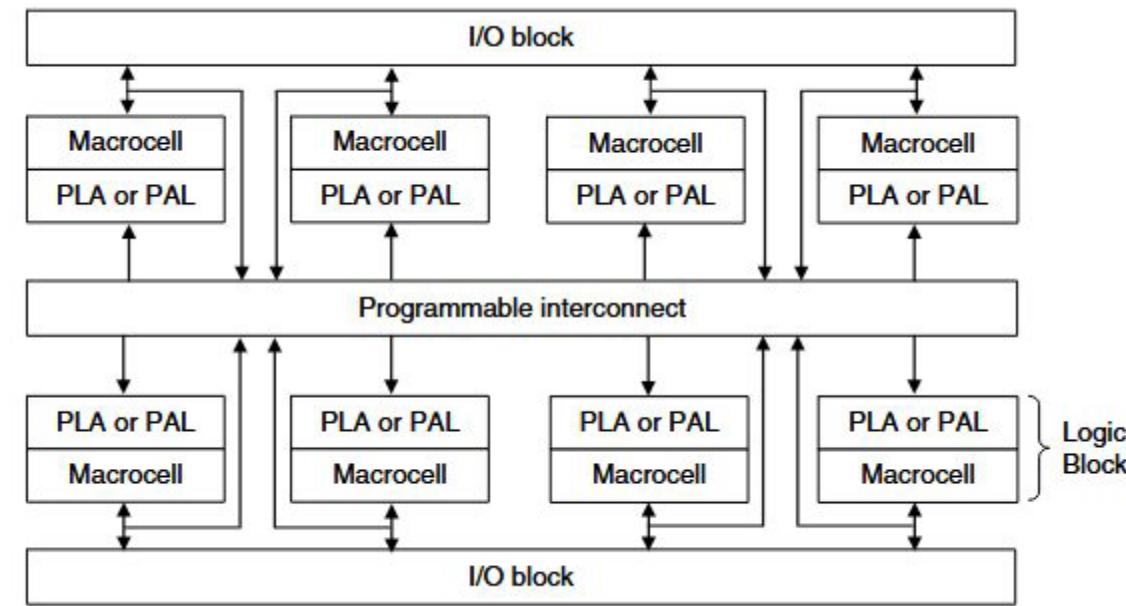
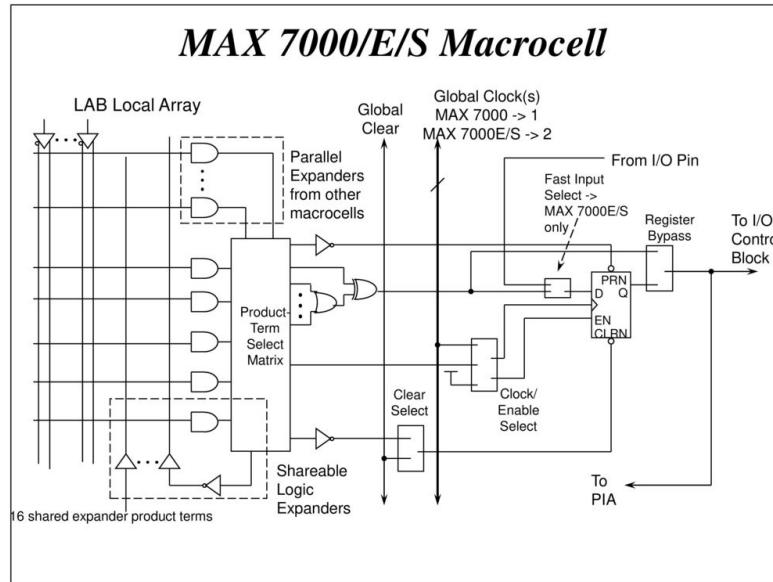
- Evolved from PALs (Programmable Array Logic), GALs (Generic Array Logic) and CPLD (Complex Programmable Logic Device)
- First generation devices (PALs) are one-time programmable; used to create combinational logical function
- Engineer writes logic expressions and defines I/Os on computer
- Software simplifies expression to SOP form and generates programming file
- Programming cable is used to set fuses / anti-fuses
- No memory / feedback elements limits applications
- Non-volatile (maintains design on power cycle)
- Using specialized machines, PAL devices were “field-programmable”
 - "One-time programmable" (OTP) devices
 - UV erasable devices
 - Flash erasable devices



Simplified programmable logic device

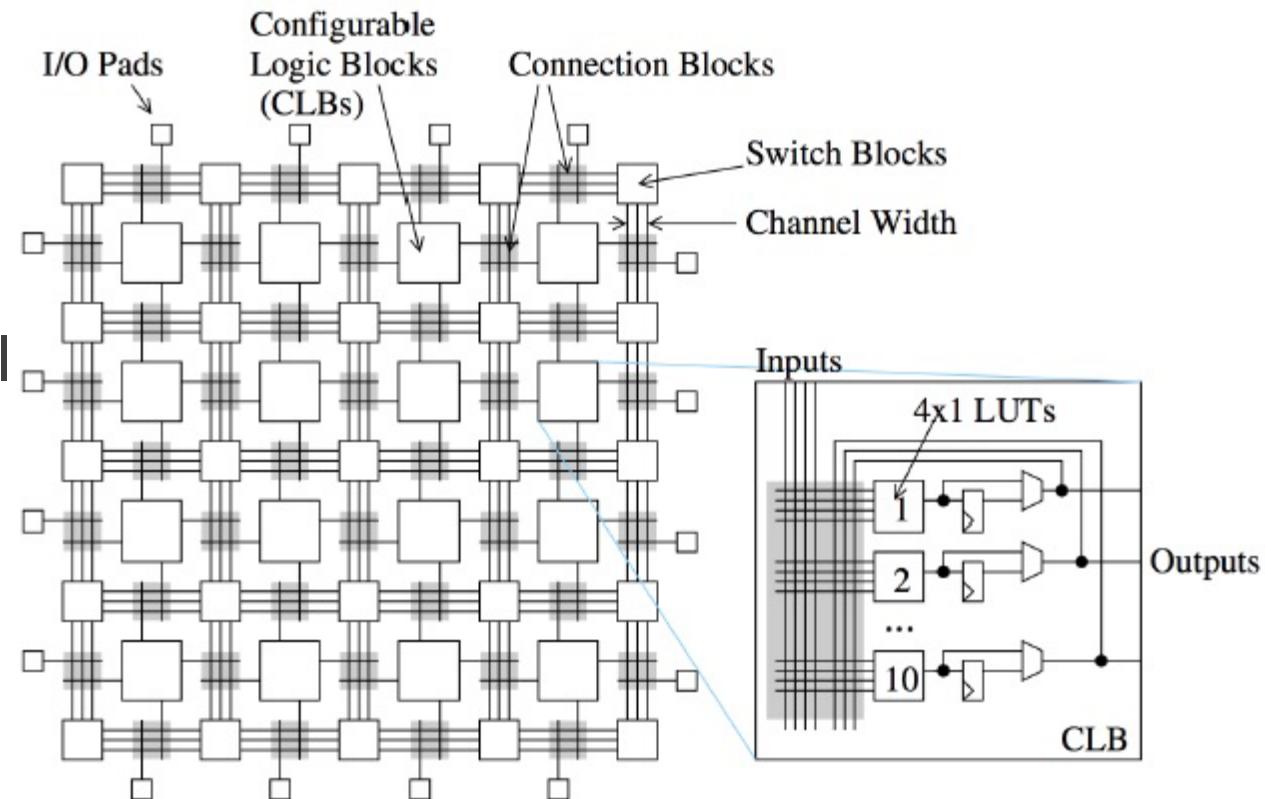
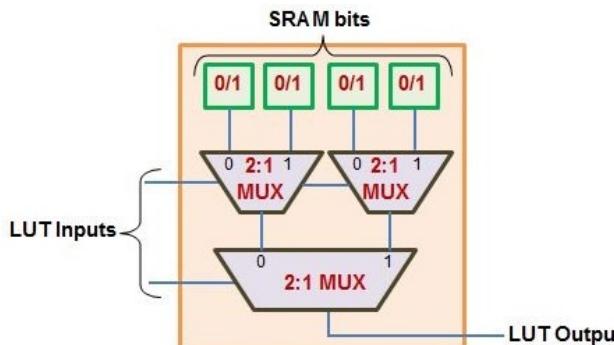
Complex Programmable Logic Devices (CPLD)

- CPLDs are one step more advanced than PAL/GAL
- combination of a fully programmable AND/OR array and a bank of macrocells
 - AND/OR array: reprogrammable, perform a multitude of logic functions
 - Macrocells: functional blocks, perform combinatorial or sequential logic
- Typically used for simple designs with small number of logic elements with limited internal routing (feedback)
- Predictable delay due to limited routing options, low density by today standards
- Non-volatile (maintains design on power cycle)



Field-Programmable Gate Array (FPGA)

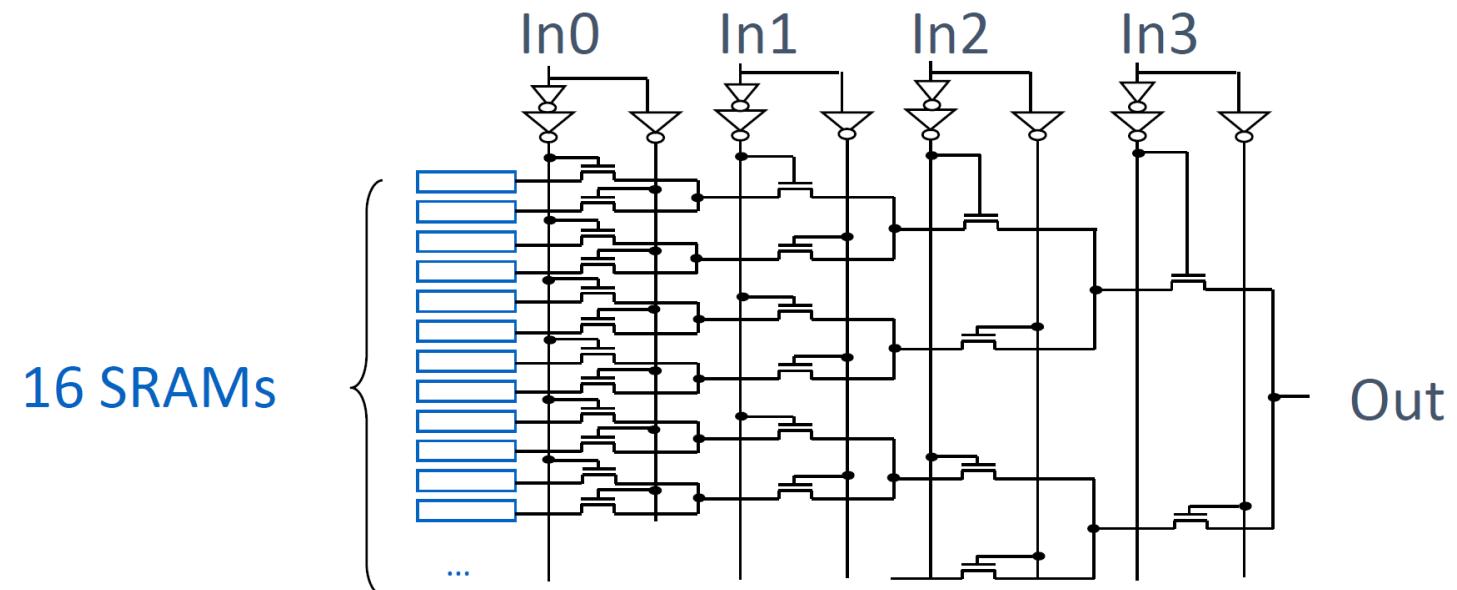
- High-density programmable gate array
- Fully programmable through bit-stream configuration file
 - Programmable Logic
 - Programmable I/O
 - Programmable Interconnects (routing)
- Look-up table (LUT) based combinational logic
 - Can be implemented with SRAM (volatile)



Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

Look-Up Table (LUT) Based Combinational Logic

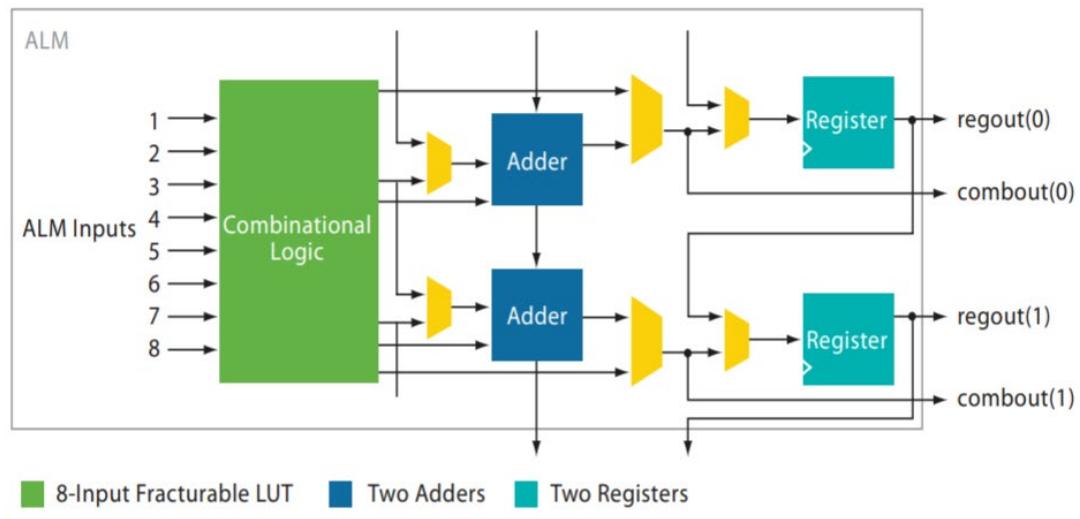
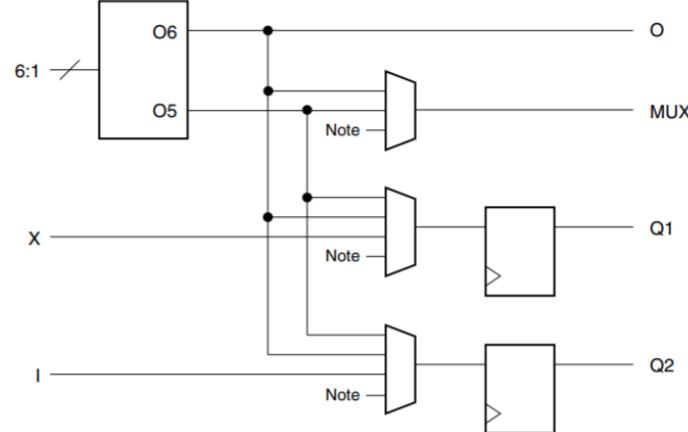
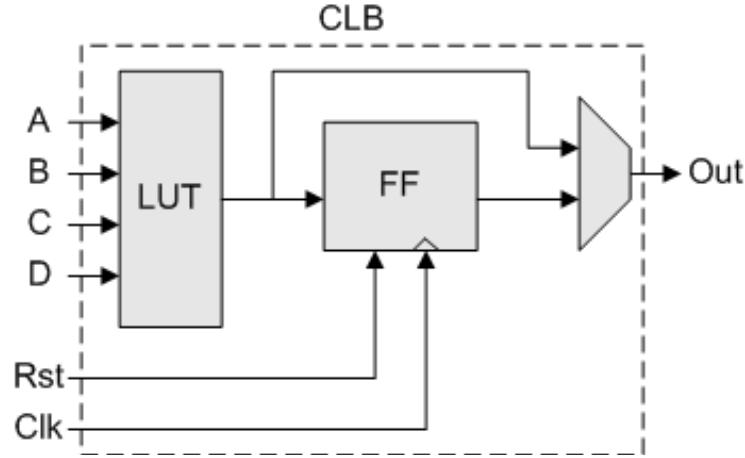
- FPGAs use LUTs (look-up tables) to express combinational logic
- Can be implemented with SRAM for high performance (this means FPGA is volatile, must be programmed on boot-up)
- E.g., 2:1 MUX can be directly implemented in single 4-input LUT (1 unused input)
- 4 input LUT is simply a 16x1 RAM



$$Out = f(in_0, in_1, in_2, in_3)$$

Configurable Logic Block (or Logic Element)

- Contains LUTs, flip-flops, MUX, etc.



Altera (Intel) Adaptive Logic Module (ALM)

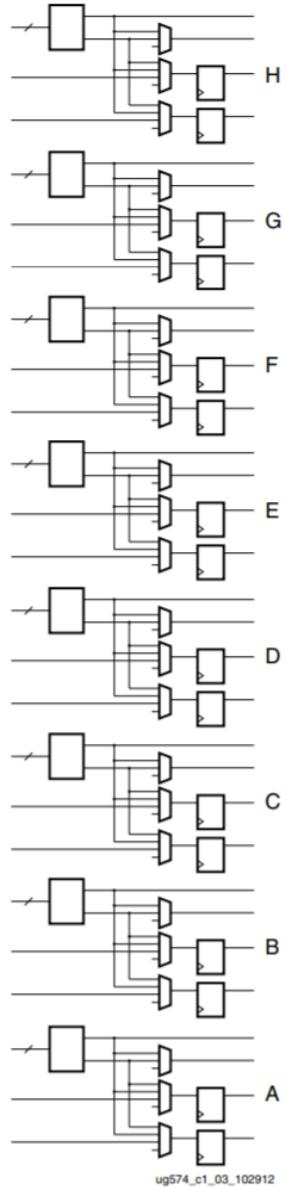
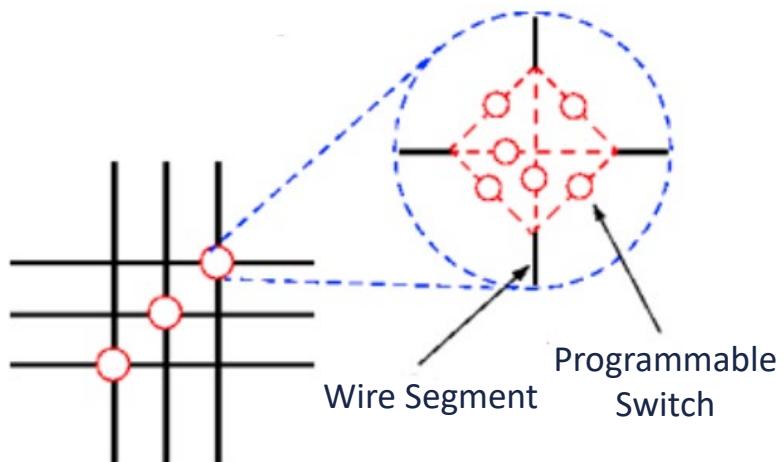
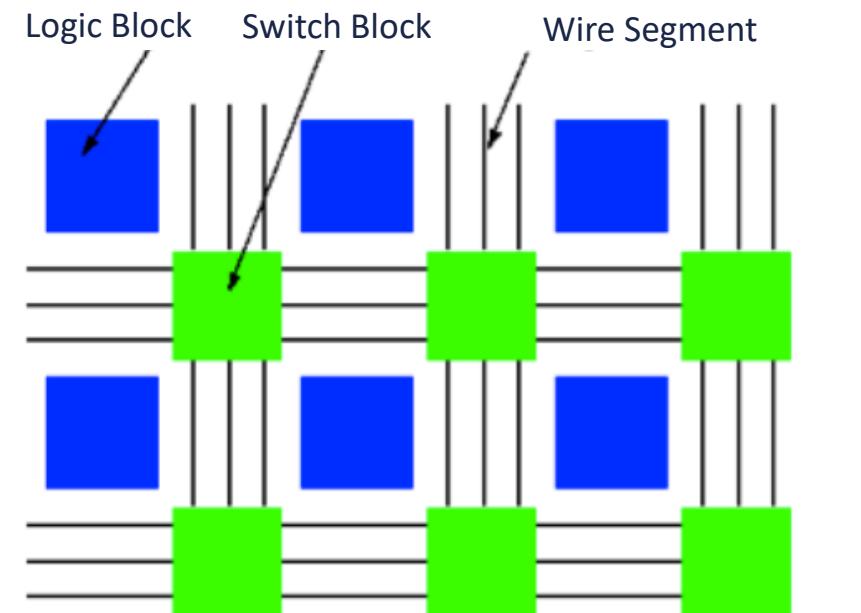
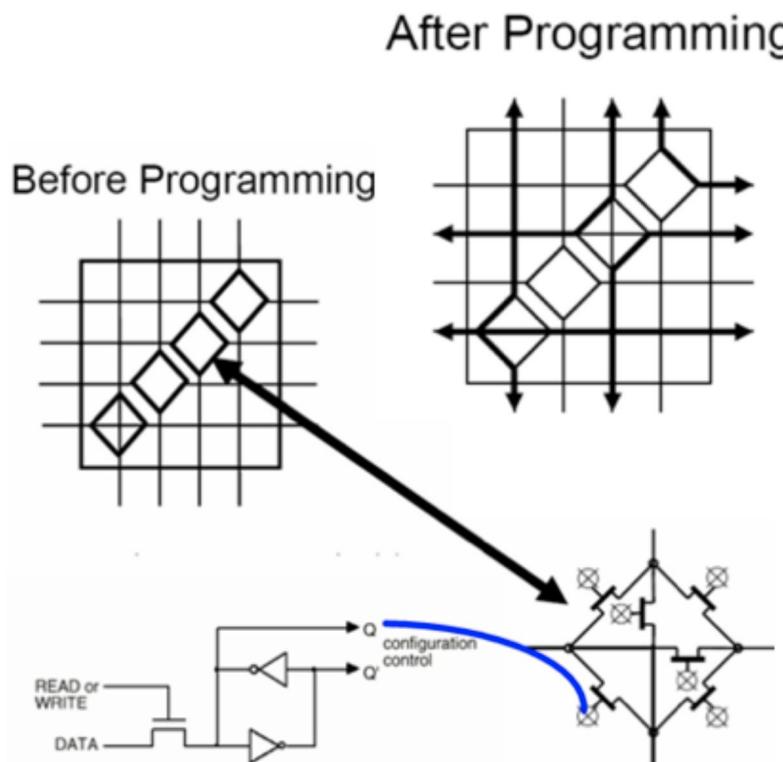


Figure 1-3: LUTs and Storage Elements in One Slice

Xilinx UltraScale CLB Slice ⁷

Programmable Routing

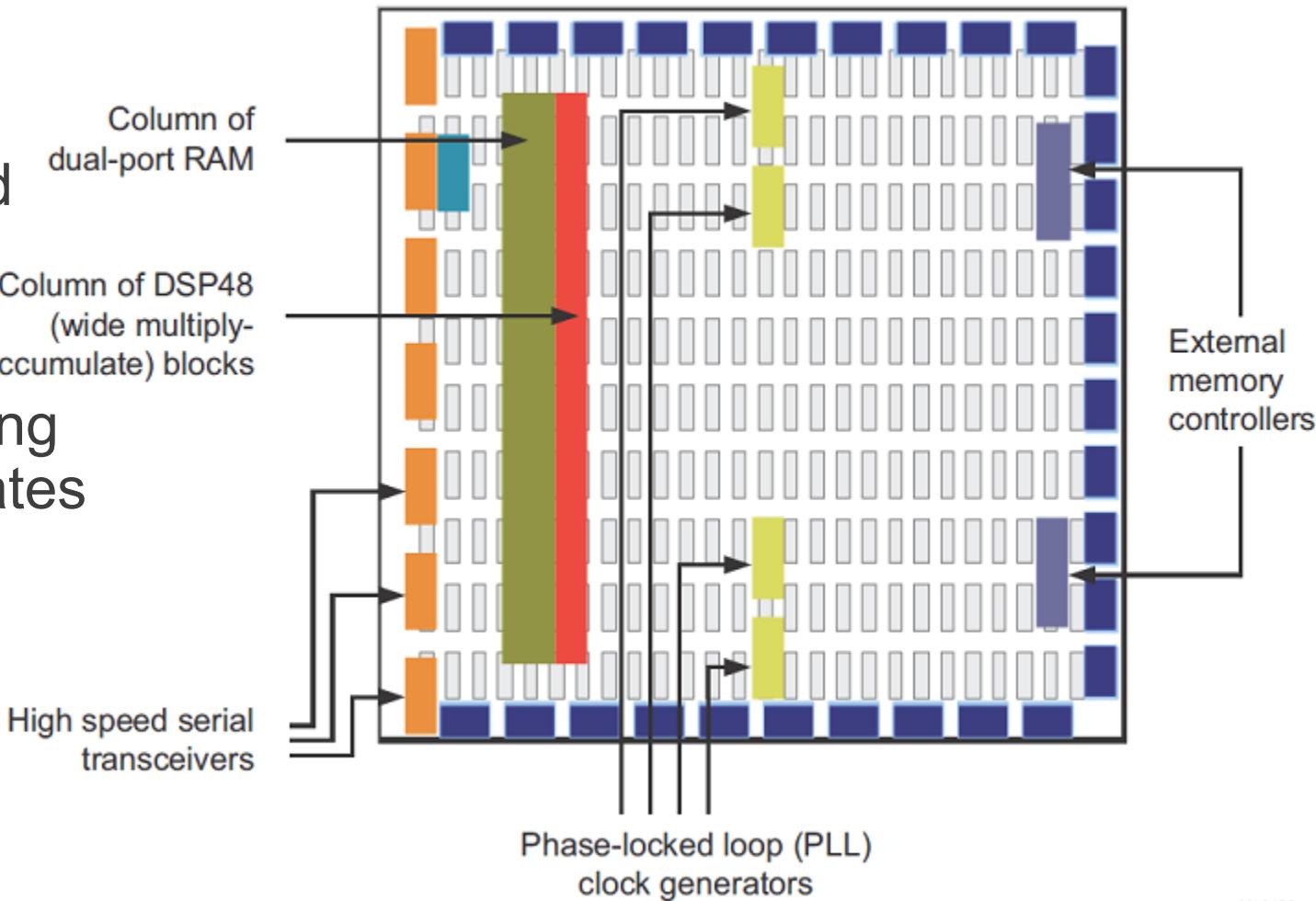
- Between rows and columns of logic blocks are wiring channels
- Programmable: a logic block pin can be connected to one of many wiring tracks through a programmable switch



Other Hardware Resources in FPGAs

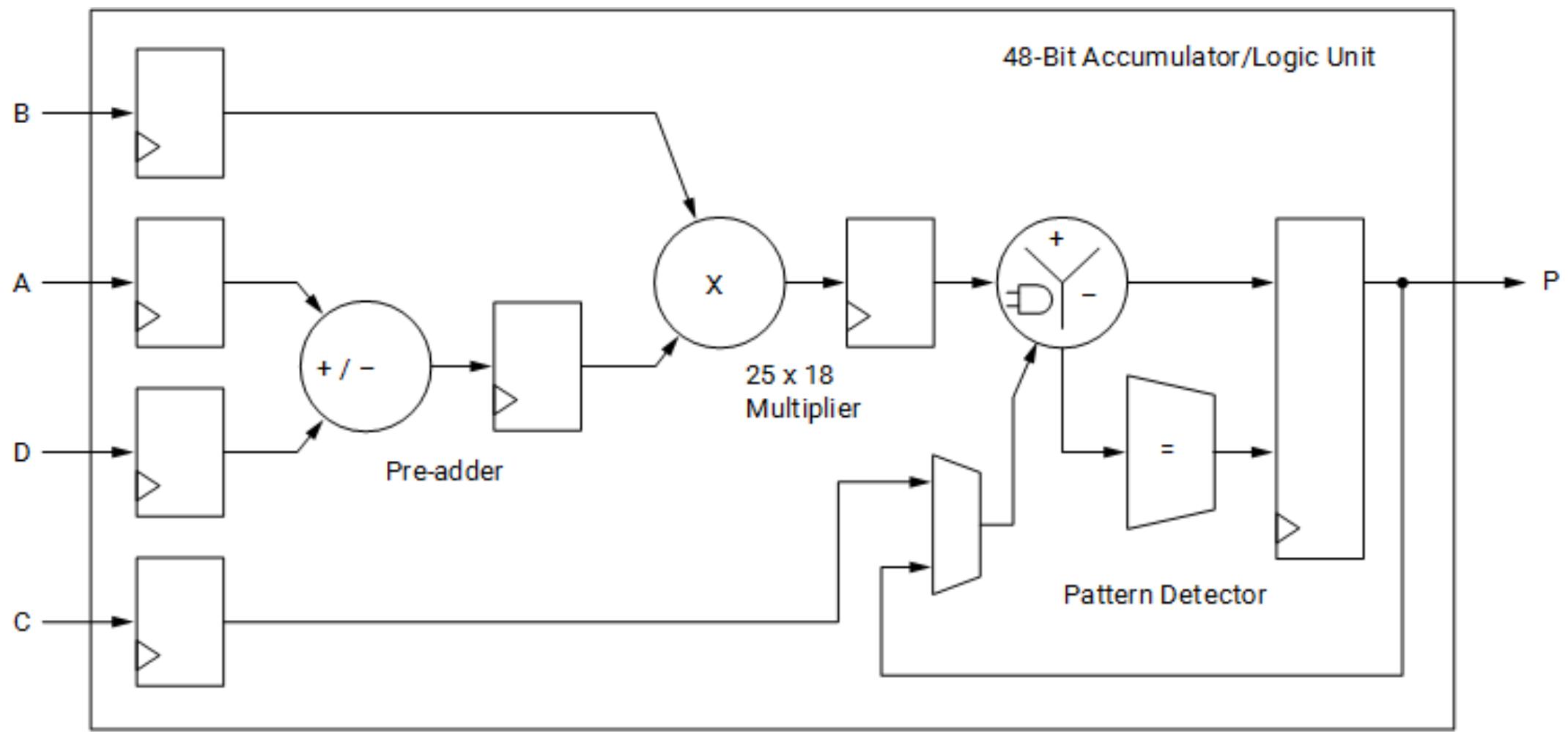
FPGAs have additional components

- High-speed serial transceivers
- Embedded memories for distributed data storage
- DSP blocks
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- Off-chip memory controllers
- PCIe controllers
- ...



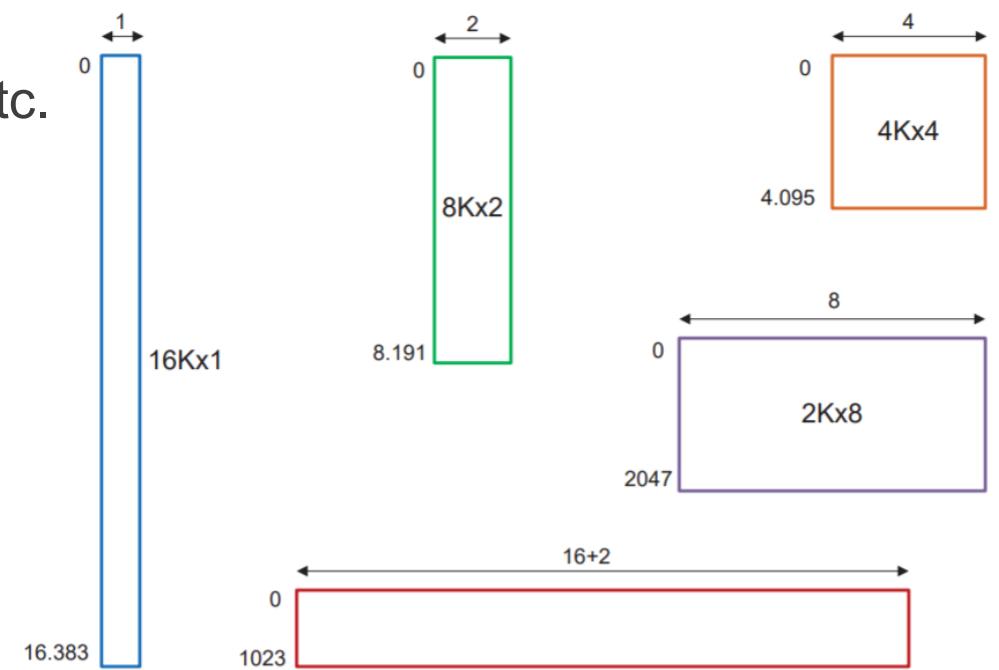
DSP48 Block in a Xilinx FPGA

- Can be used to implement " $P = B \times (A + D) + C$ " or " $P += B \times (A + D)$ "

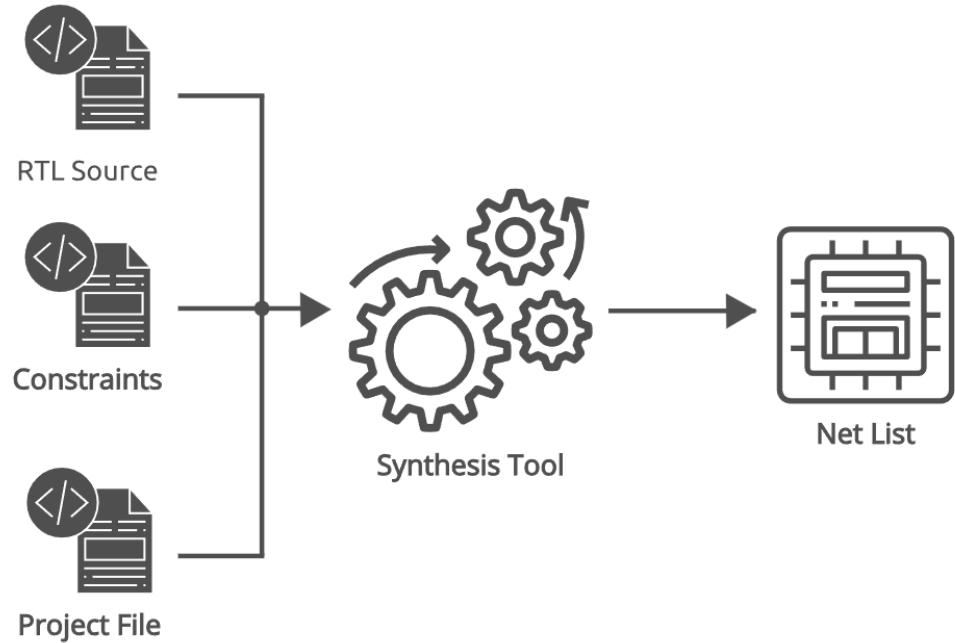
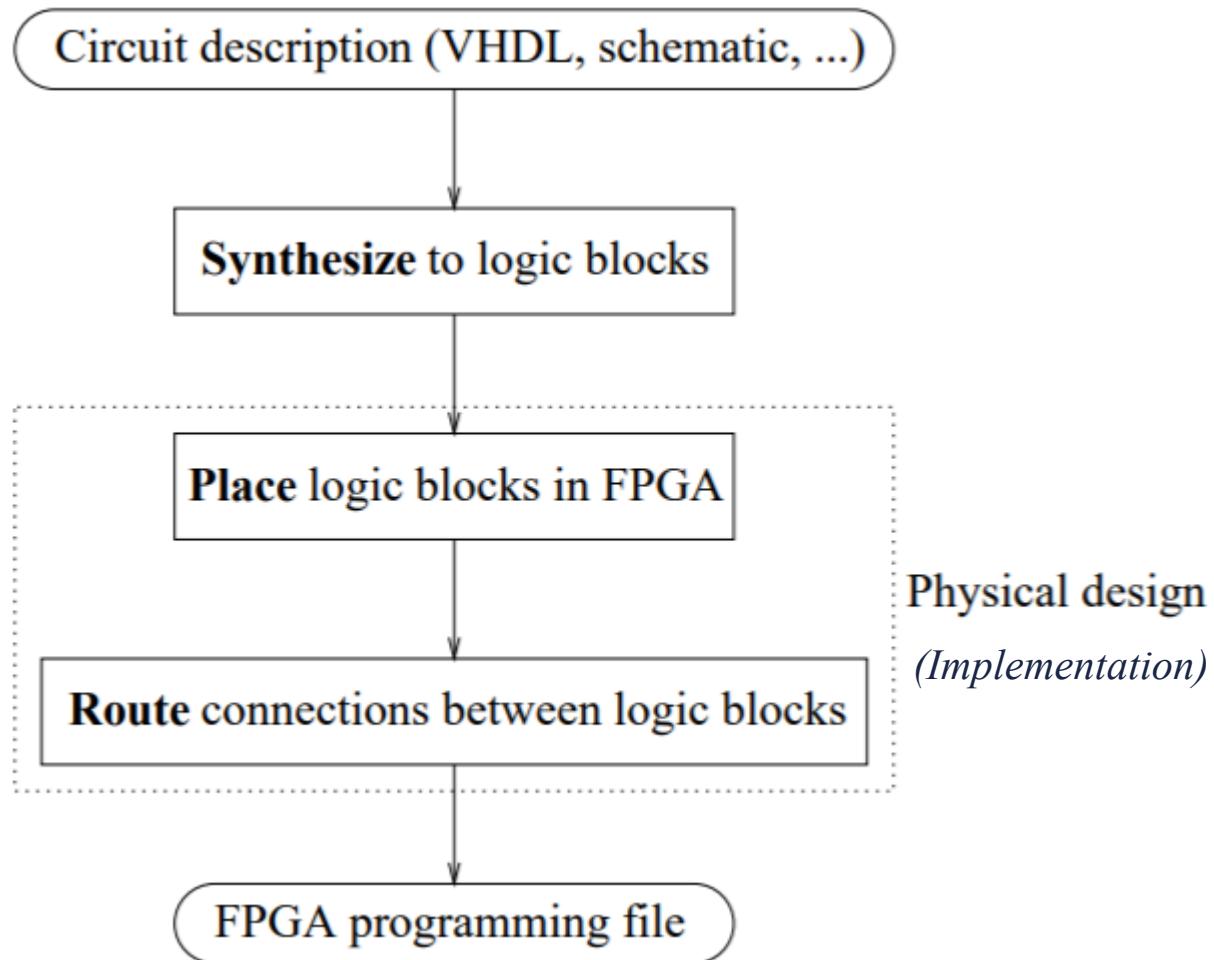


Block RAM (BRAM)

- BRAM is a dual-port RAM module instantiated into the FPGA fabric
- Can hold either 18k or 36k bits
 - Can be configured into different sizes
- Multiple configuration options
 - True dual-port, simple dual-port, single-port, etc.
- Two independent ports
 - Individual address, clock, write enable, clock enable, etc.
 - Independent widths for each port



FPGA Synthesis



Hardware Description Language (HDL)

- Specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits
- Two major languages: Verilog and VHDL
- Different types of description: dataflow, behavioral, and structural

Dataflow

```
module example1 ( e , a, b, do, d1,
d2, d3);
    input e, a, b;
    output do, d1, d2, d3;
    assign d0 = ( e & ~a & ~b); //00
    assign d1 = (e & ~a & b);   //01
    assign d2 = (e & a & ~b);   //10
    assign d3 = ( e & a & b);   //11
endmodule
```

Behavioral

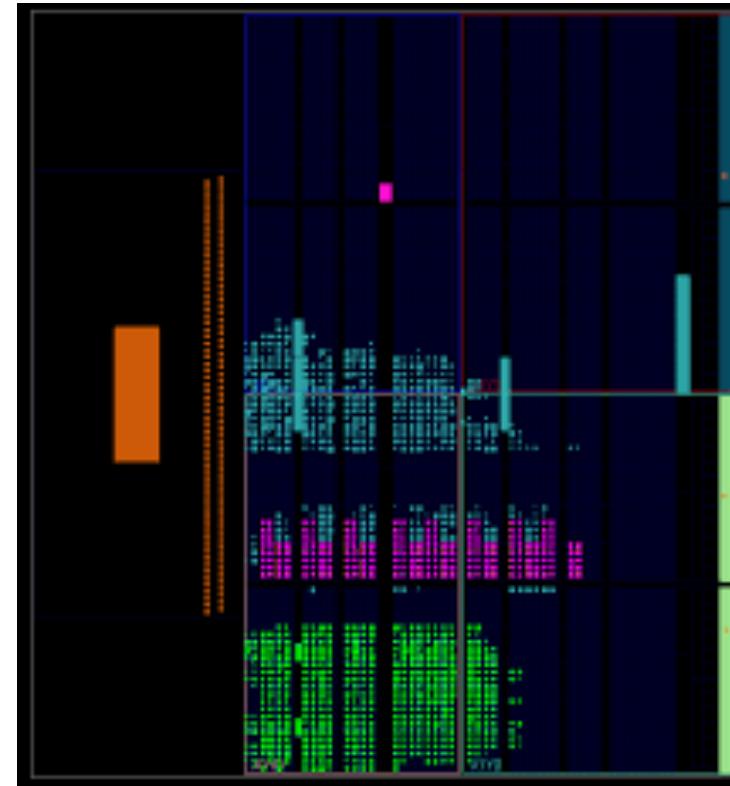
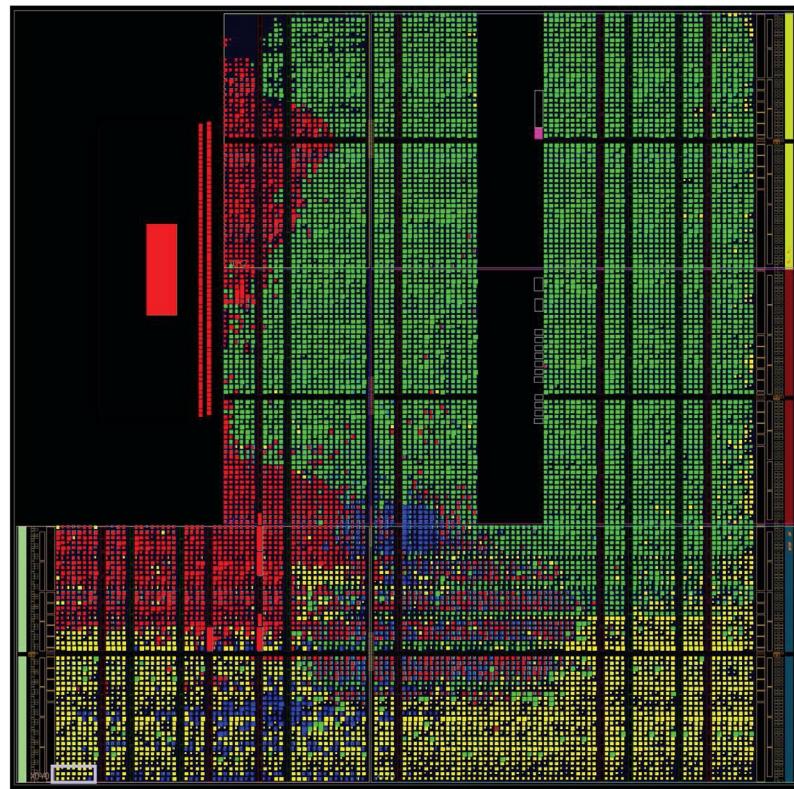
```
module example2 (e, i, d);
    output [3:0] d;
    input [1:0] i;
    input e;
    reg [3:0] d;
    always @ (i or e) begin
        if (e==1) begin
            case (i)
                0: d = 4'b 0001;
                1: d = 4'b 0010;
                2: d = 4'b 0100;
                3: d = 4'b 1000;
                default d = 4'b xxxx;
            endcase
        end
        else
            d = 4'b0000;
    end
endmodule
```

Structural

```
module build_xor (a, b, c);
    input a, b;
    output c;
    wire c, a_not, b_not;
    not a_inv (a_not, a);
    not b_inv (b_not, b);
    and a1 (x, a_not, b);
    and a2 (y, b_not, a);
    or out (c, x, y);
endmodule
```

Configuring the FPGA

- Programming an FPGA: download a *bitstream* (not a program, but a configuration) to an FPGA
- All FPGA components (logic blocks, interconnects, etc.) are configured using the bitstream so that they implement the correct functionalities



Reading Materials

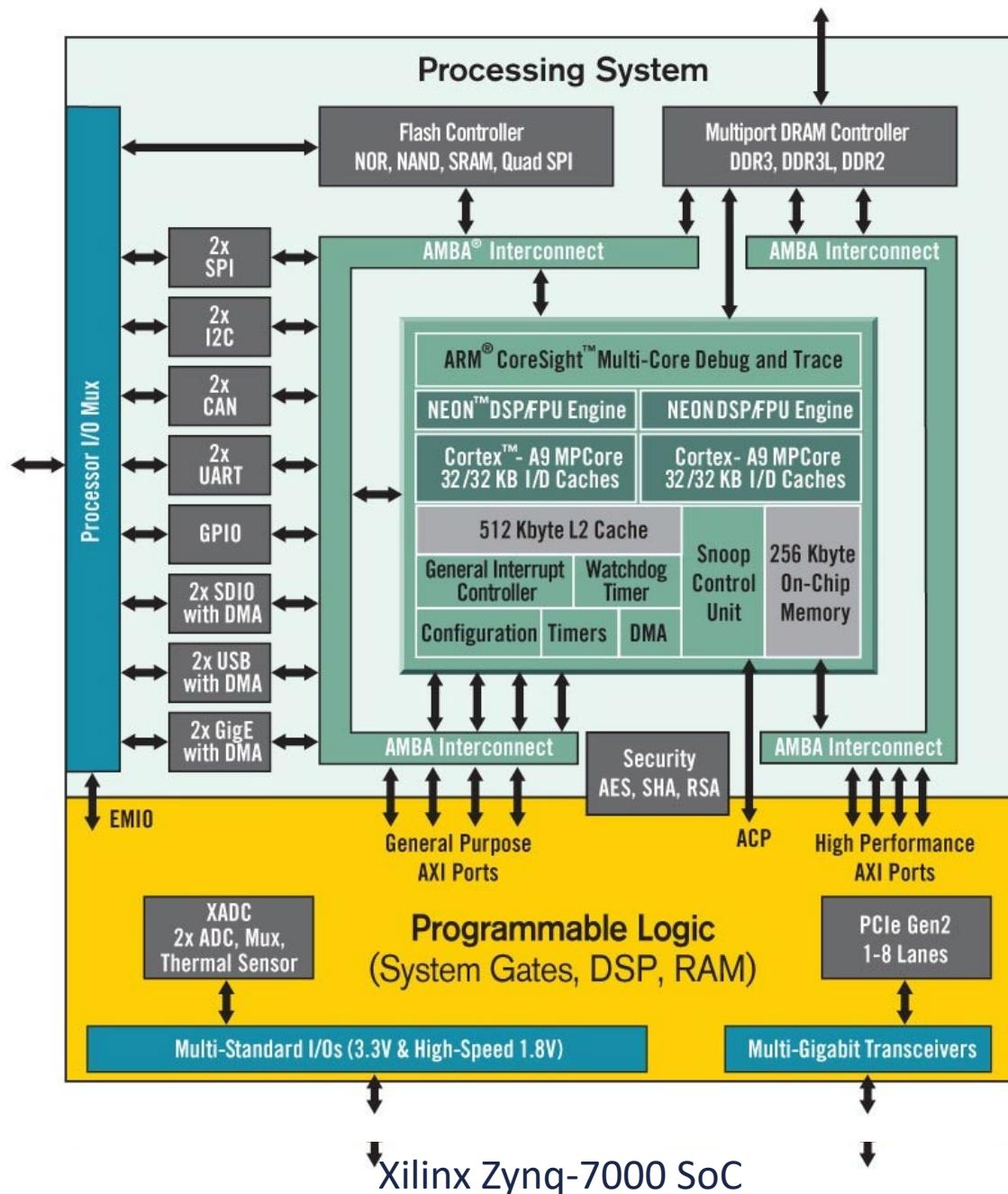
- Andrew Boutros and Vaughn Betz, FPGA Architecture: Principles and Progression. 2021. [\[PDF\]](#)
- FPGA Architecture Basics. https://www.rapidwright.io/docs/FPGA_Architecture.html

Xilinx Zynq SoC Overview

- SoC can be divided into two parts:
 - **PS** : Processing System (ARM CPU)
 - **PL**: Programmable Logic (FPGA)

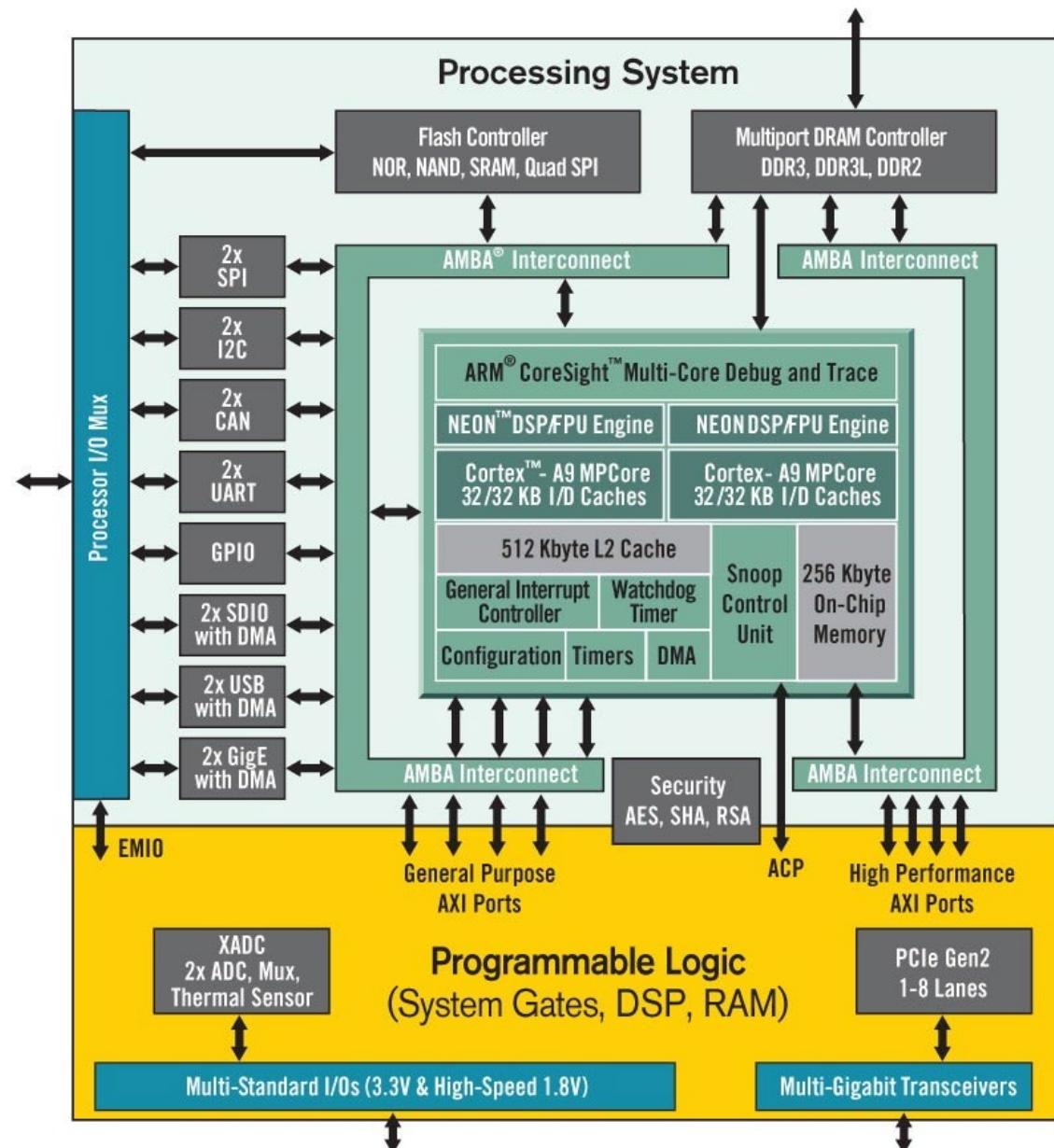
Note (for Xilinx Zynq-7000 SoC on the right):

- No dedicated video controller or GPU
- Peripherals can be connected to PS or PL



Xilinx Zynq SoC Overview

- PS: Dual Core ARM A9
 - 1-2 GOPS
- Floating point support
 - ARM NEON support as well
- Up to 1GHz Clock
- L2 Cache
 - Unified 512KB
- AXI High performance interconnects
- 256KB on-chip memory scratchpad
- PL: FPGA
 - 85K Logic cells + 220 DSP slices
 - 10-100 GOPS!



Zynq SoC Boot Flow

Multi-Stage

1. Run from ROM

- a) Copy FSBL from boot device to OCM (on-chip memory)

2. First Stage Boot Loader (FSBL)

- a) Load Uboot from boot device to DDR
- b) Program PL
- c) Initiate PS boot

3. Uboot

- a) Load linux kernel to DDR
- b) Device tree init
- c) FPGA init

4. OS Boot

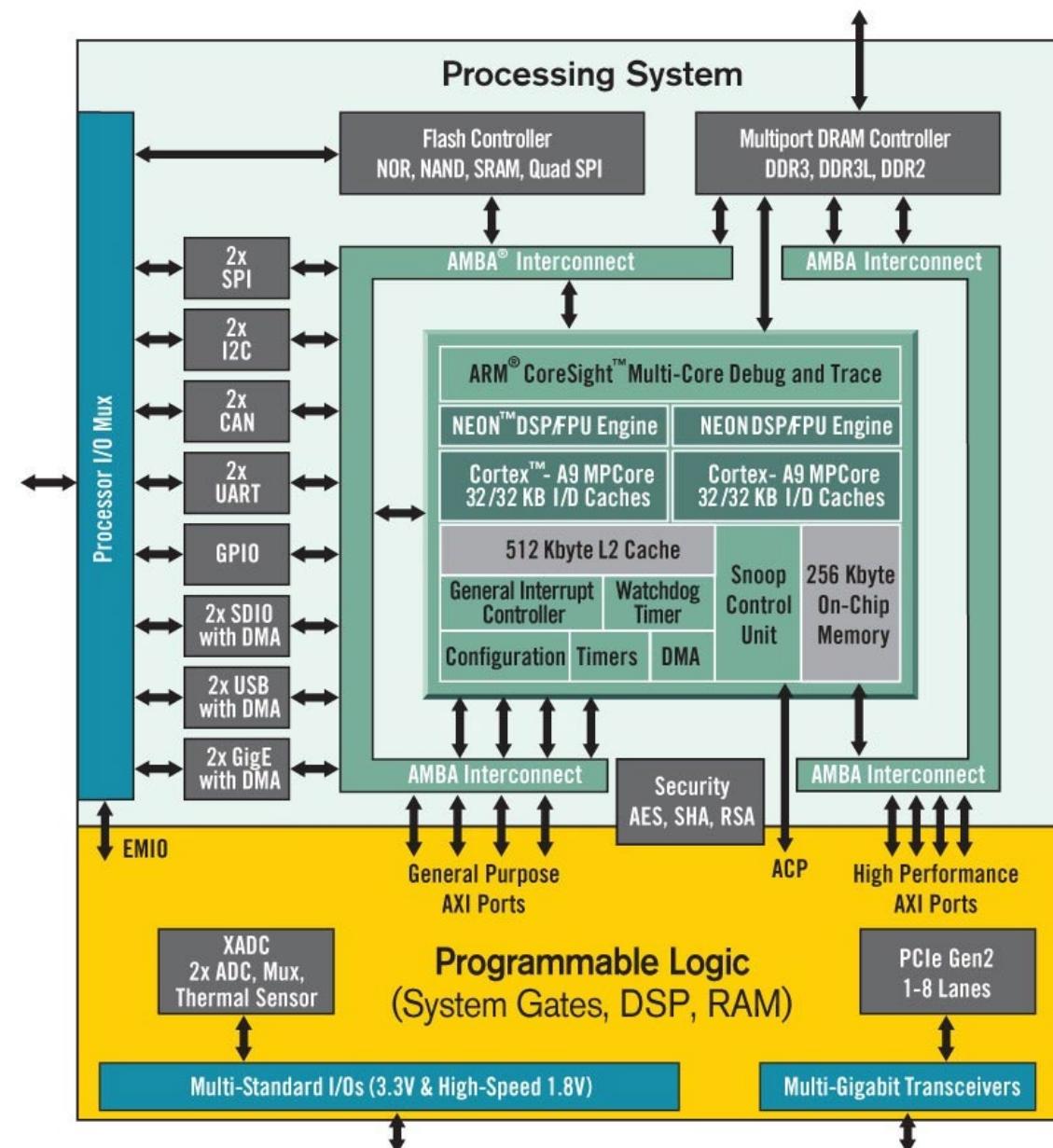
- a) Linux Boot

More Info:

<http://www.wiki.xilinx.com/Getting+Started>
<http://xillybus.com/tutorials/device-tree-zynq-1>

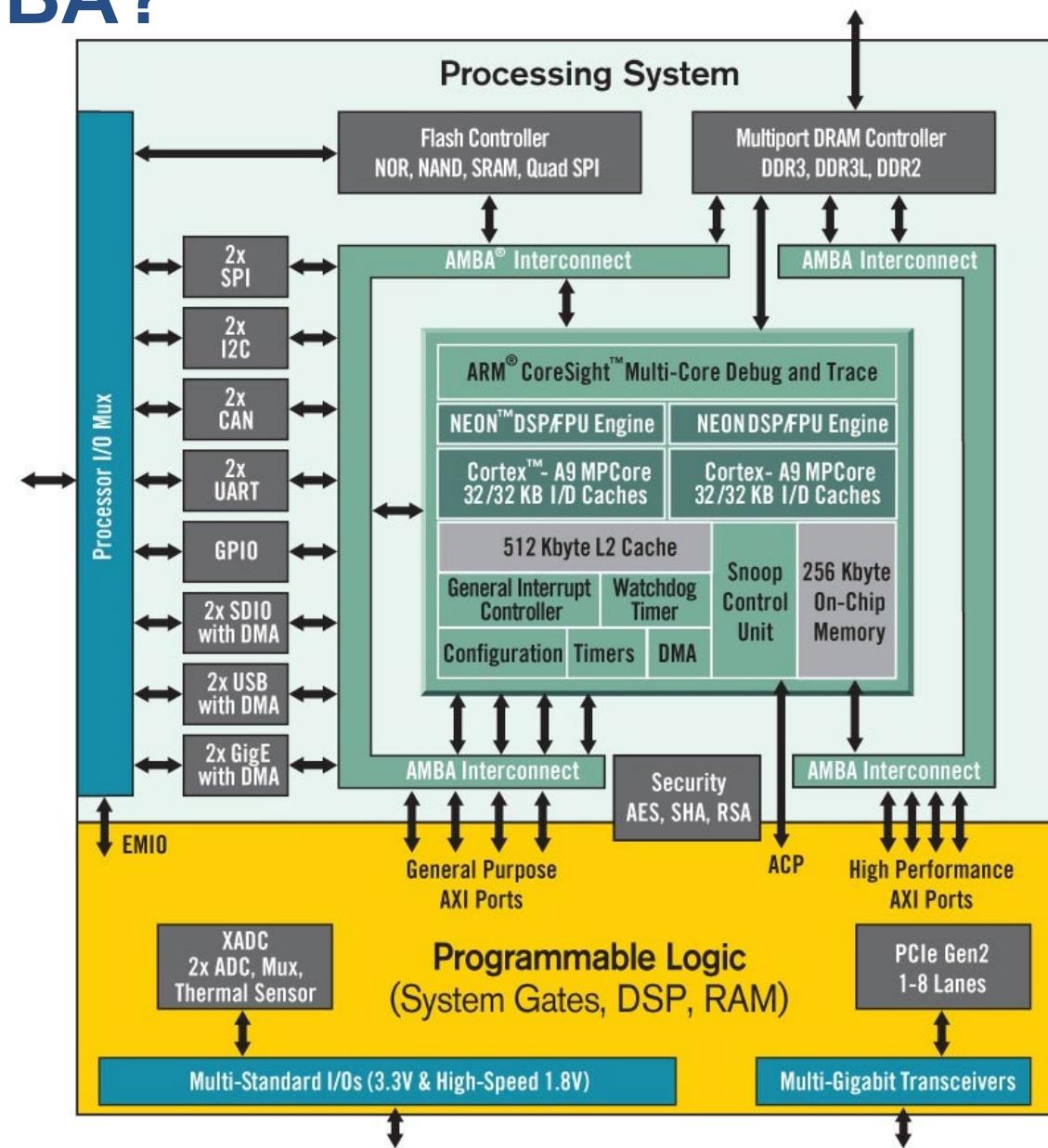
Xilinx Zynq SoC: Interrupts

- Several interrupts available for PS-PL interface
- 16 peripheral interrupts available from PL to PS
 - Use for accelerator dev
- PS to PL interrupts exists as well
 - Read up via manual



Xilinx Zynq SoC: What is AXI/AMBA?

- AMBA: Advanced Microcontroller Bus Architecture
 - Protocol
 - Open standard, on-chip interconnect for SoC
- AXI:
 - Advanced eXtensible Interface
 - Very common AMBA interface
- Why:
 - Flexible
 - IP reuse
 - Ease of use



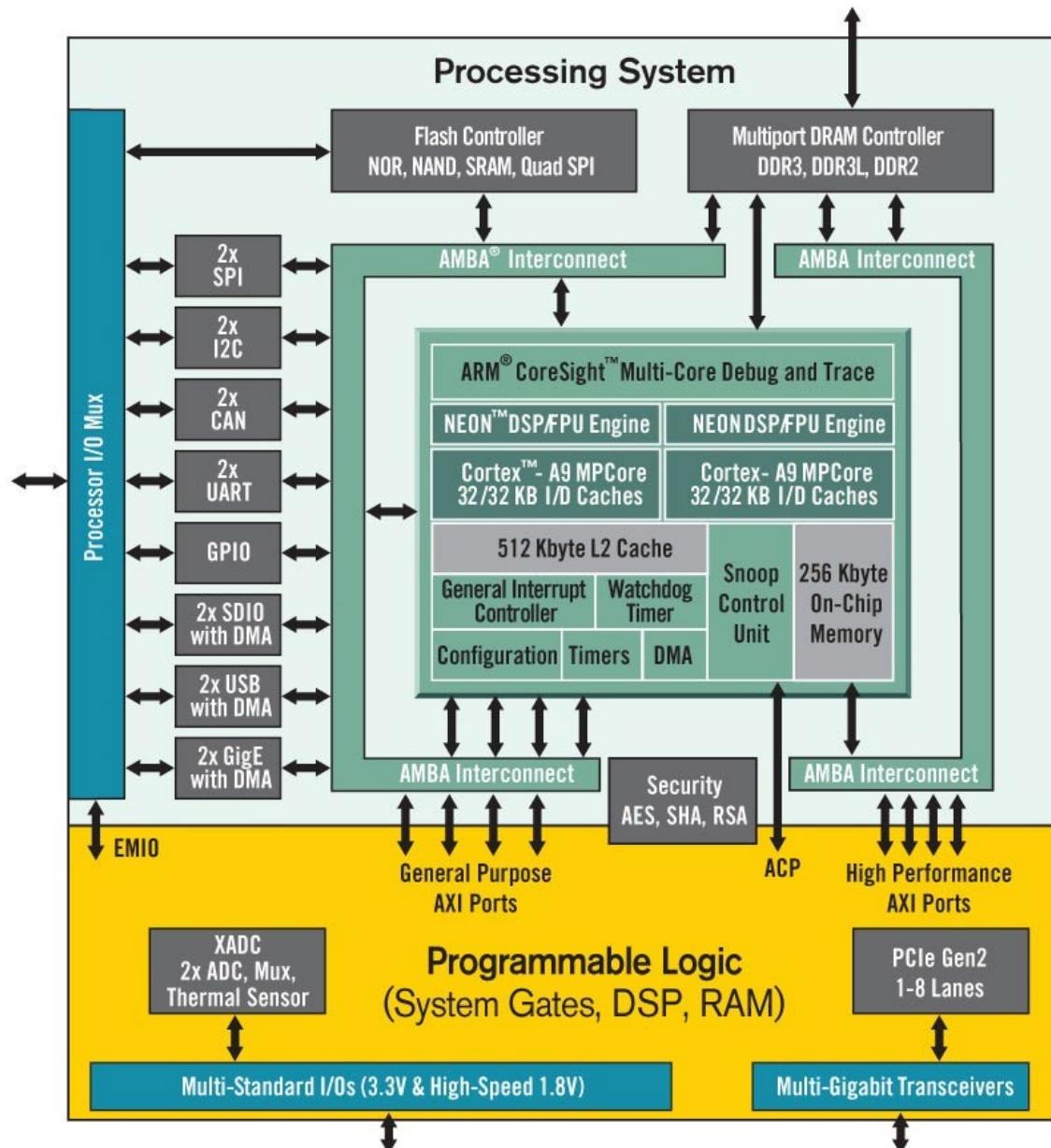
Xilinx Zynq SoC: AXI

AXI Interfaces:

- AXI4 *Mainly for Data Movement*
 - High performance
 - Memory mapped
- AXI4-Lite *Mainly for Control and Status*
 - Low throughput
 - Memory mapped
- AXI4-Stream *Mainly for Data Movement*
 - High performance
 - Streaming Data

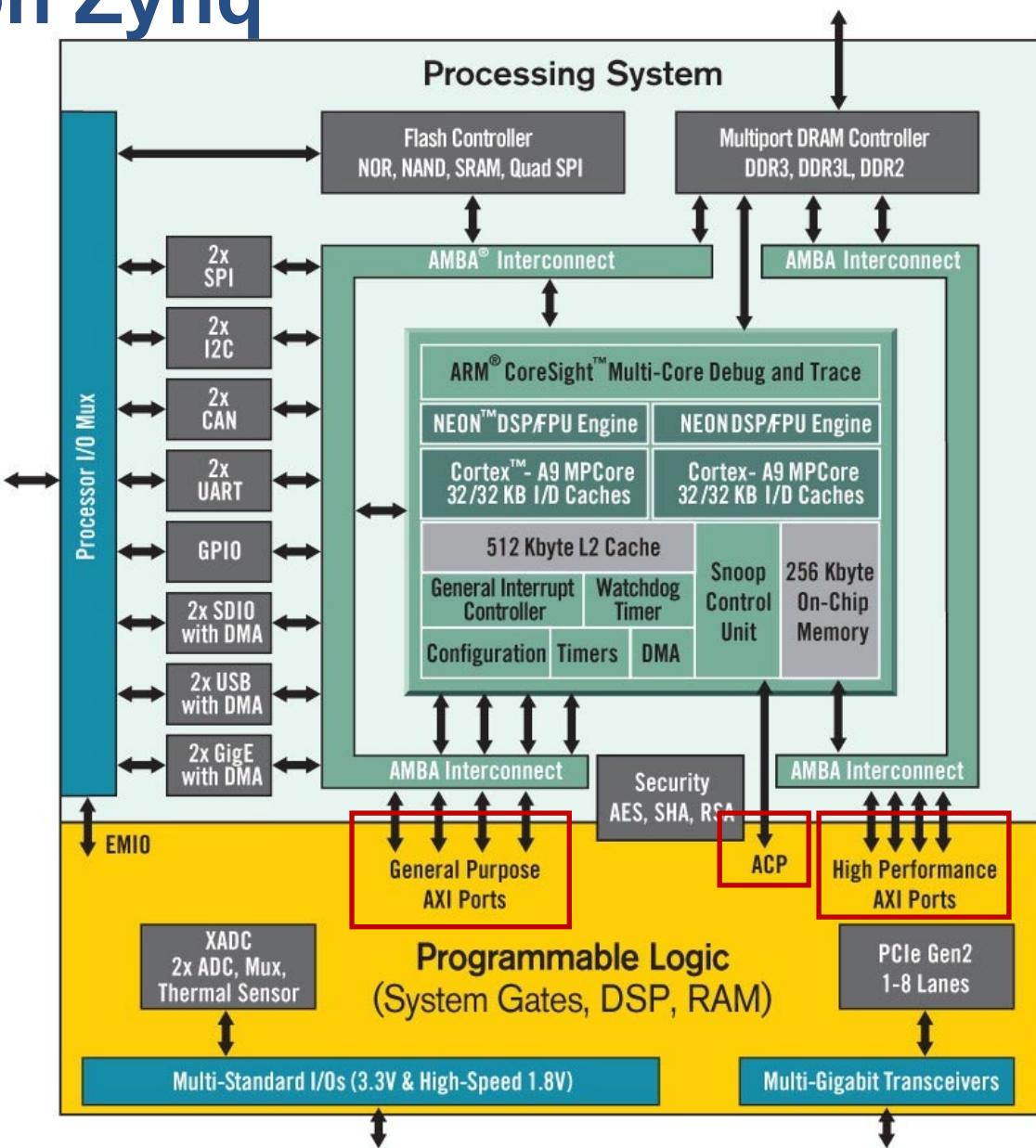
More Info:

http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

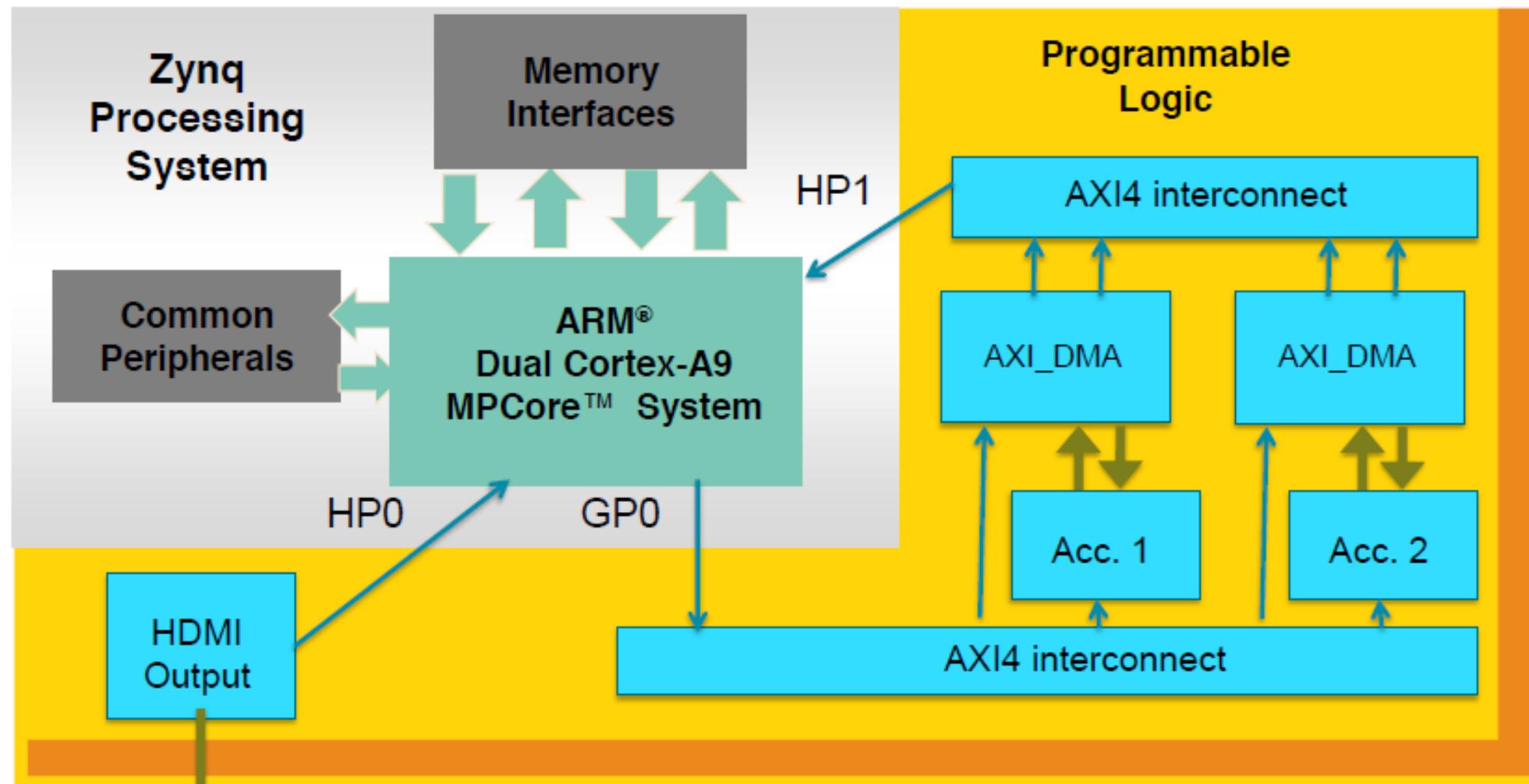


Xilinx Zynq SoC: AXI Interfaces on Zynq

- HP (High Performance): **Mainly for burst data movement**
 - 4x64bit Slave
 - **High bandwidth** access to external memory
 - @150MHz, bandwidth = 9.6GB/s
 - Large Data bursts
- GP (General Purpose): **Mainly for Control and Status**
 - 2x32bit Slave
 - PL to PS peripherals
 - 2x32bit Master
 - PS to PL access
- ACP (Accelerator Coherency Port): **for cache-coherent data access**
 - 1x64bit Slave
 - PL accesses processor L2 cache
 - Hardware coherence

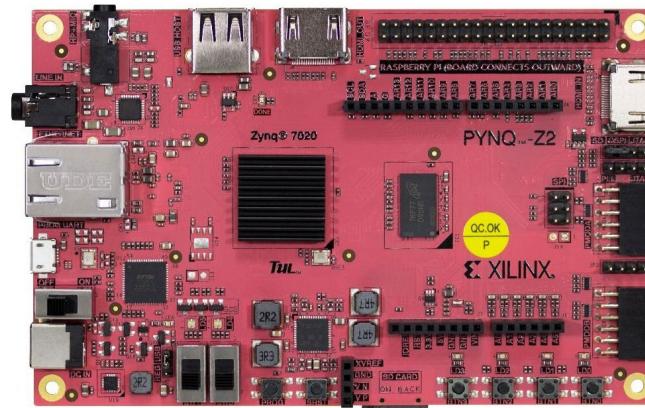


Xilinx Zynq SoC: Example



PYNQ Boards

- PYNQ-Z2 contains a Zynq-7000 SoC
 - Zynq SoC: ZYNQ XC7Z020-1CLG400C
 - 650MHz dual-core Cortex-A9 processor
 - Programmable logic equivalent to Artix-7 FPGA
 - 13,300 logic slices
 - 630 KB of fast block RAM
 - 220 DSP slices
 - Memory
 - 512MB DDR3 with 16-bit bus @ 1050Mbps
 - 16MB Quad-SPI Flash
 - MicroSD Slot
 - USB and Ethernet
 - Gigabit Ethernet PHY
 - Micro USB-JTAG Programming circuitry
 - Micro USB-UART bridge
 - USB 2.0 OTG PHY (supports host only)
- Audio and Video (input/output)
- Switches, Push-buttons and LEDs
- Expansion Connectors
 - Two standard Pmod ports
 - 16 total FPGA I/O
 - Arduino connector
 - Raspberry Pi connector



Supports PYNQ programming environment

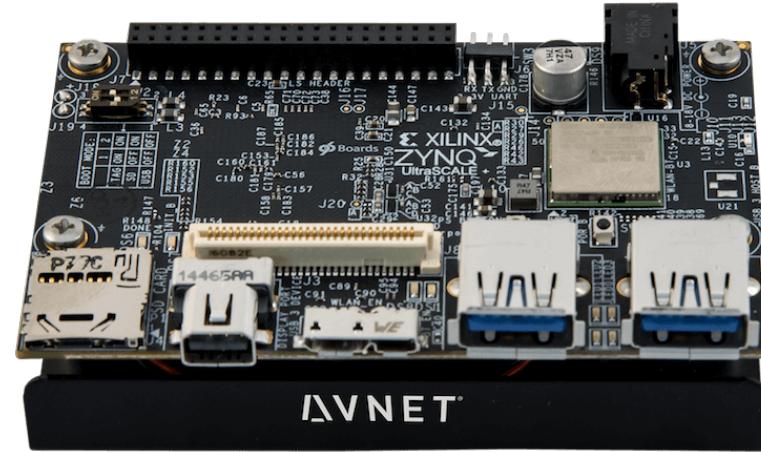
PYNQ-Z2 specs: https://www.tul.com.tw/images/PYNQ-Z2_PA_v2_pp_20201209_STD.pdf

PYNQ-Z2 user manual: https://dpoauwgwqsy2x.cloudfront.net/Download/PYNQ_Z2_User_Manual_v1.1.pdf

PYNQ-Z2 setup guide: https://pynq.readthedocs.io/en/v2.6.1/getting_started/pynq_z2_setup.html

Ultra96 Boards

- Ultra96 contains a Zynq UltraScale+ MPSoC
 - MPSoC: Xilinx Zynq UltraScale+ MPSoC ZU3EG A484
 - Memory: Micron 2GB LPDDR4 memory
 - Storage: Delkin 16GB microSD card + adaptor
 - Wireless: 802.11b/g/n Wi-Fi and Bluetooth 4.2
 - USB and Ethernet
 - 1x USB 3.0 Type Micro-B upstream port
 - 2x USB 3.0, 1x USB 2.0 Type A downstream ports
 - Display: Mini DisplayPort (MiniDP or mDP)
 - Expansion interface:
 - 40-pin 96Boards Low-speed expansion header
 - 60-pin 96Boards High speed expansion header



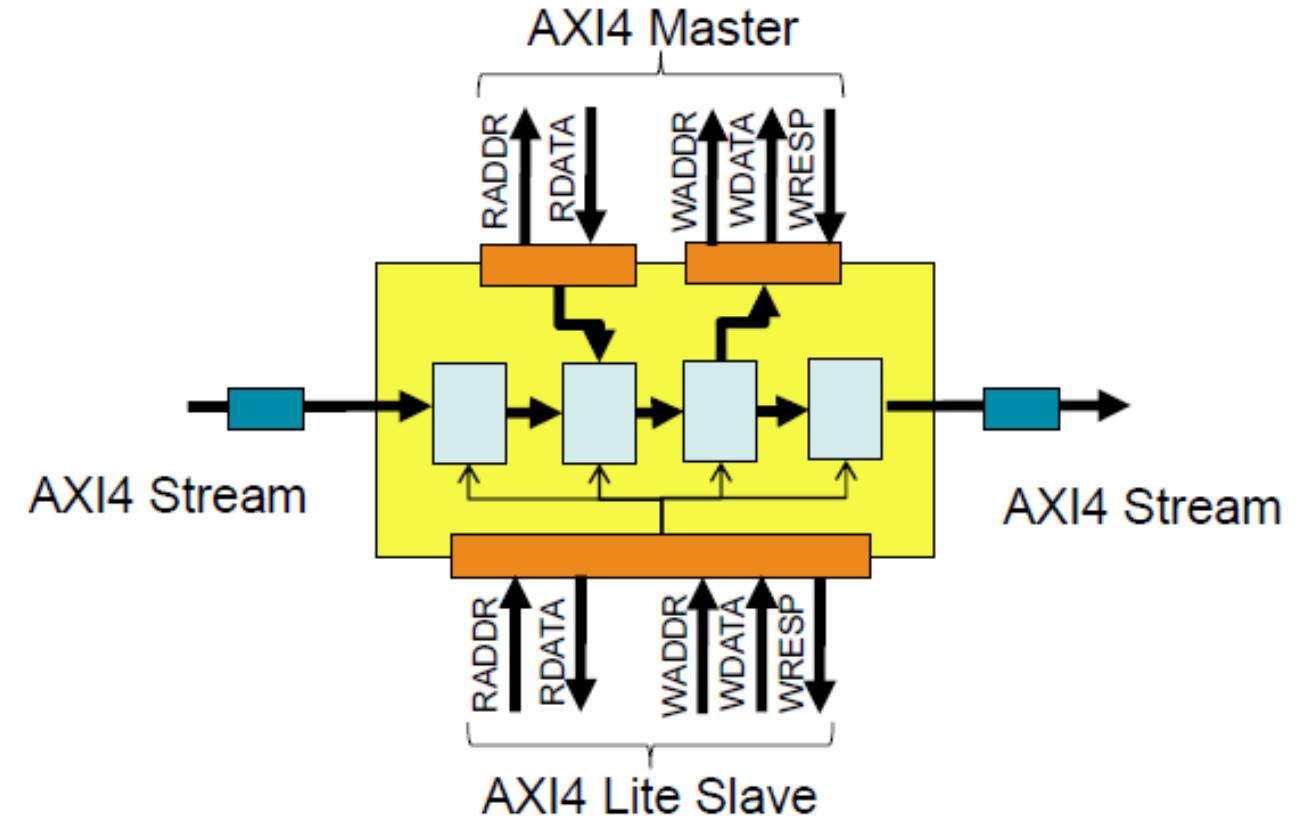
Supports PYNQ programming environment

Ultra96-v2 user's guide: https://www.element14.com/community/servlet/JiveServlet/downloadBody/92688-102-2-395912/Ultra96-V2-HW-User-Guide-v1_1.pdf

Ultra96-v2 PYNQ setup guide: https://ultra96-pynq.readthedocs.io/en/latest/getting_started.html

Zynq SoC: Accelerator Development Example

- Example:
 - Design a vector add accelerator to perform $c[i] = (a[i] + b[i])/N$ for given vectors
 - Variable size arrays
 - Performance and power
 - Arrays to initialized in software
 - N is constant
- Choose Interface:
 - How should I move data to FPGA?
 - Stream or memory-mapped?
- Latency calculations
 - Time to process/move data
- Resource Usage
 - How much parallelism is available?
- On-FPGA memory
 - Can all vectors be stored on FPGA?
- HW-SW co-design



Zynq SoC: Accelerator Development Example

- Simplify Example (sequential Vector Add)

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

- Basic Performance modeling/estimation:

- Break down into smaller operations
- Compute time per operation
- Eg. Read latency = r , Write latency = w , floating point add latency = c

- Vector Add
 - Repeat N times
 - LOAD A[i]
 - LOAD B[i]
 - ADD C[i], A[i], B[i]
 - STORE C[i]

$$\text{Total time} = N * (2r + c + w)$$

Zynq SoC: Accelerator Development Example

Total time = $N * (2r + c + w)$

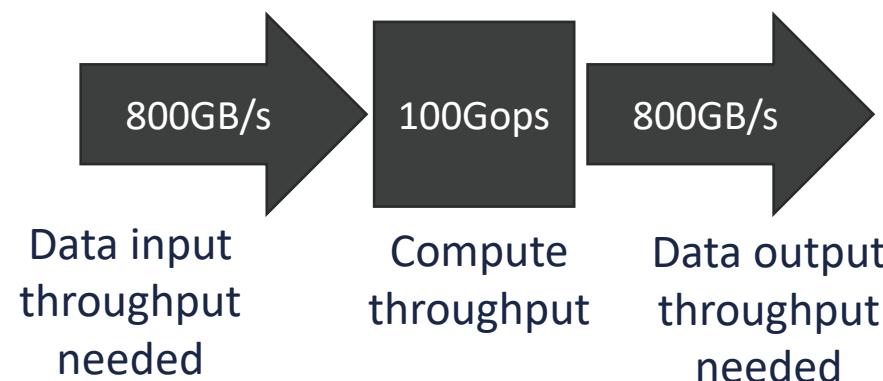
- What does this tell us?
 - How large does N need to be before it makes sense to accelerate this?
 - For each compute operation, 12 bytes of data is moved
 - How do I bring data in?
 - MEMORY < --- > PS < --- > PL
 - MEMORY < --- > PL

- Bandwidth vs Latency:

- Bandwidth:
How much data can you bring in per unit time.
- Latency:
How long does it take data to arrive.
- To perform K operations in parallel, you need $K * 12\text{bytes}$
 - B/W needed = $K * 12 * \text{Frequency}$
 - Choose K wisely

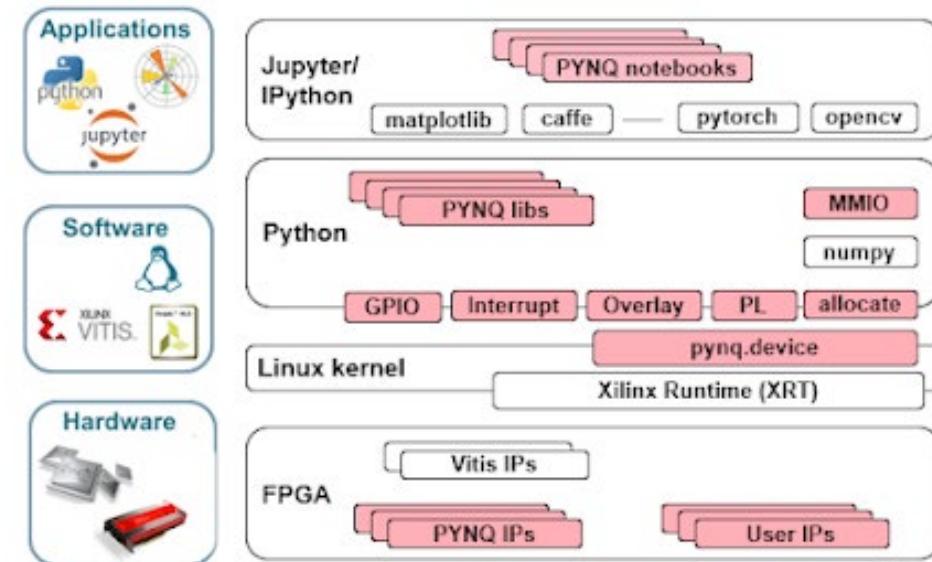
Zynq SoC: Accelerator Development Example

- Is 100GOPS really possible?
 - Yes and No
 - Assumption: Each op is floating point
 - 8 bytes per op
 - Total bandwidth needed 800Gb/s?
- Look to Data Reuse
 - On-Chip memory can be a limit
 - Streaming/Systolic Design:
 - Smaller blocks, feeding into each other
 - Be smart about bandwidth
 - Dedicated read and dedicated write channels may not be smart
 - Think about communication patterns
 - Do you need concurrent read and write?



What is PYNQ?

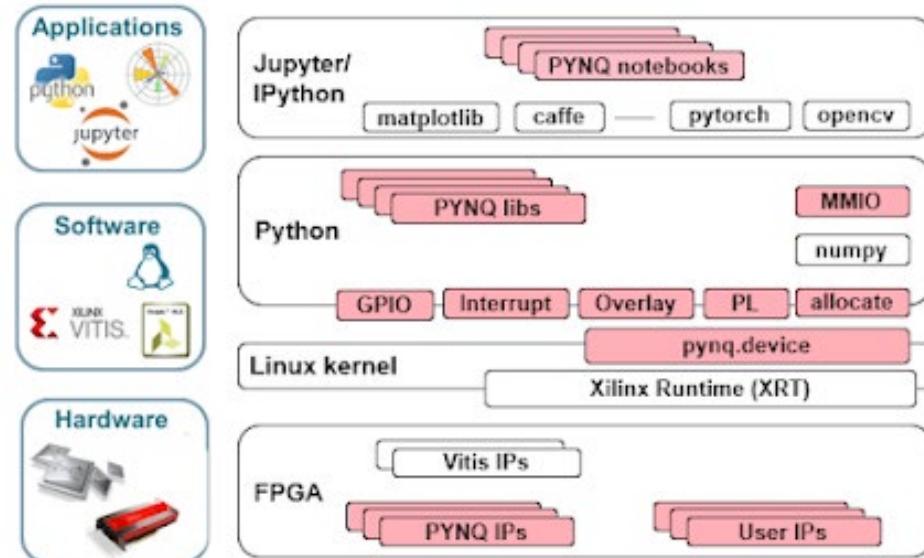
- An open-source project from Xilinx® that makes it easier to use Xilinx platforms
- **Python**-based APIs and libraries
- Simplifies **host** programming
- Supports wide range of Xilinx devices:
 - Zynq, Zynq UltraScale+, Zynq RFSoC, MPSoC, Alevo, AWS-F1, etc.



What is PYNQ?

Key Technologies

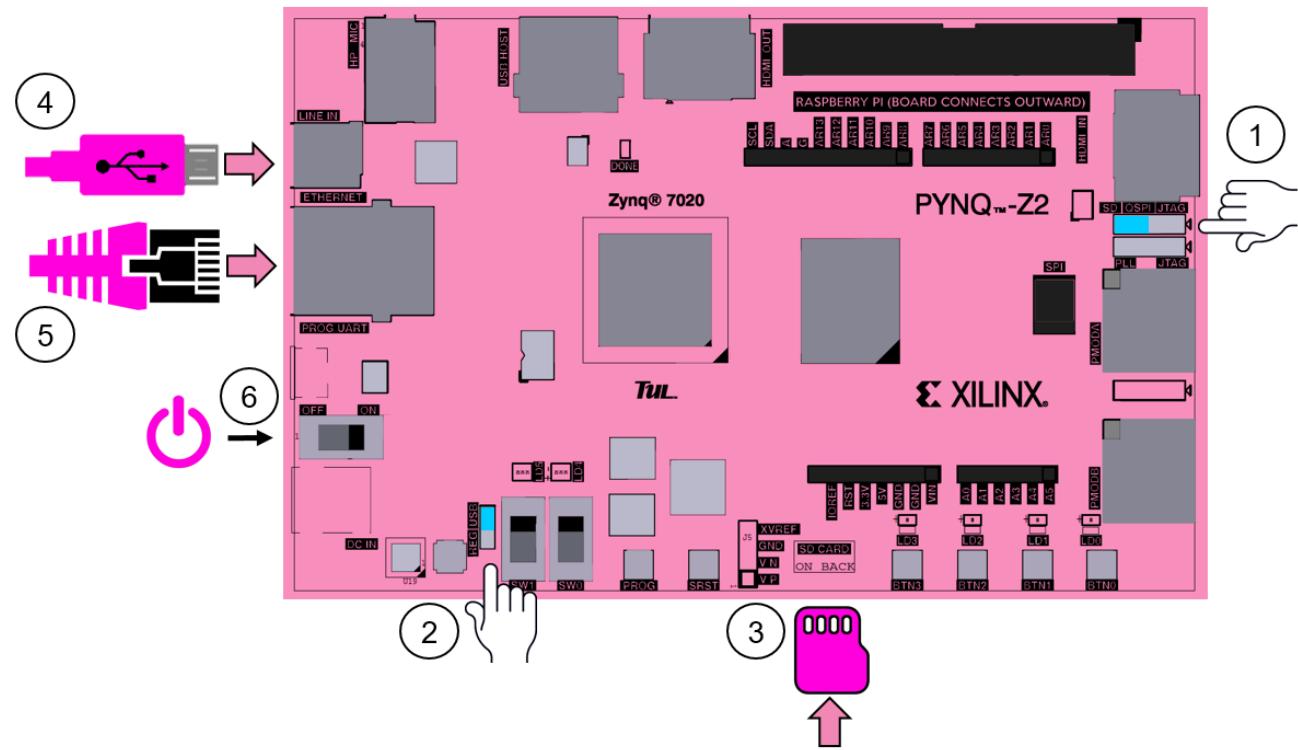
- Jupyter Notebook: a browser based interactive computing environment
- PYNQ enabled FPGA boards can be programmed in Jupyter notebook using Python (host programming)
- PYNQ is delivered in two forms:
 - Bootable Linux image for Zynq boards
 - Open-source Python package for Alveo and AWS-F1



Getting Started with PYNQ

Using PYNQ-Z2 board as example

1. Set the **Boot** jumper to the **SD** position (boot from the Micro-SD card)
2. To power the board from the micro-USB cable, set the **Power** jumper to the **USB** position. (You can also power the board from an external 12V power regulator by setting the jumper to **REG**.)
3. Insert the Micro SD card loaded with the PYNQ-Z2 image into the **Micro SD** card slot underneath the board
4. Connect the USB cable to your PC/Laptop, and to the **PROG - UART** Micro-USB port on the board
5. Connect the Ethernet port by following the instructions below
6. Turn on the PYNQ-Z2

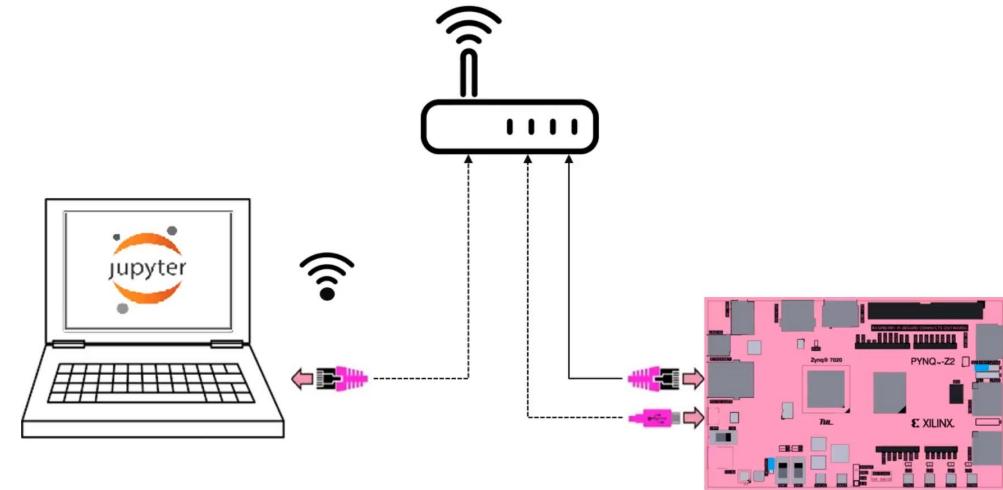


Getting Started with PYNQ

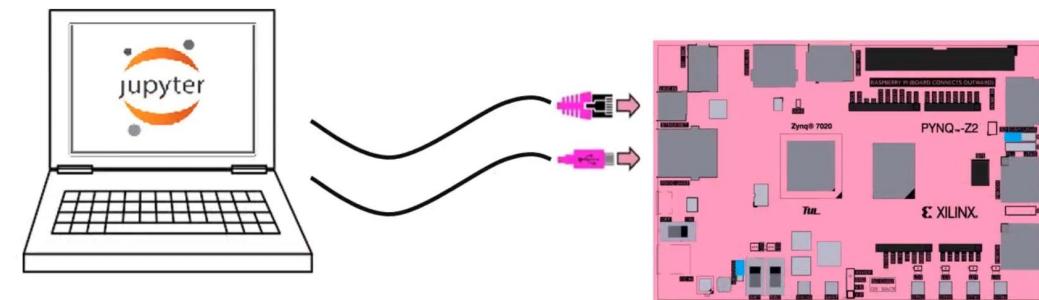
Network connection

Once your board is setup, you need to connect to it to start using Jupyter notebook

- Option A: Connect to a Network Router
 - Connect the Ethernet port on your board to a router/switch
 - Connect your computer to Ethernet or Wi-Fi on the router/switch
 - Browse to <http://<board IP address>>
- Option B: Connect to a Computer
 - Assign your computer a static IP address
 - Connect the board to your computer's Ethernet port
 - Browse to <http://192.168.2.99>



Option A: Connect to a Network Router



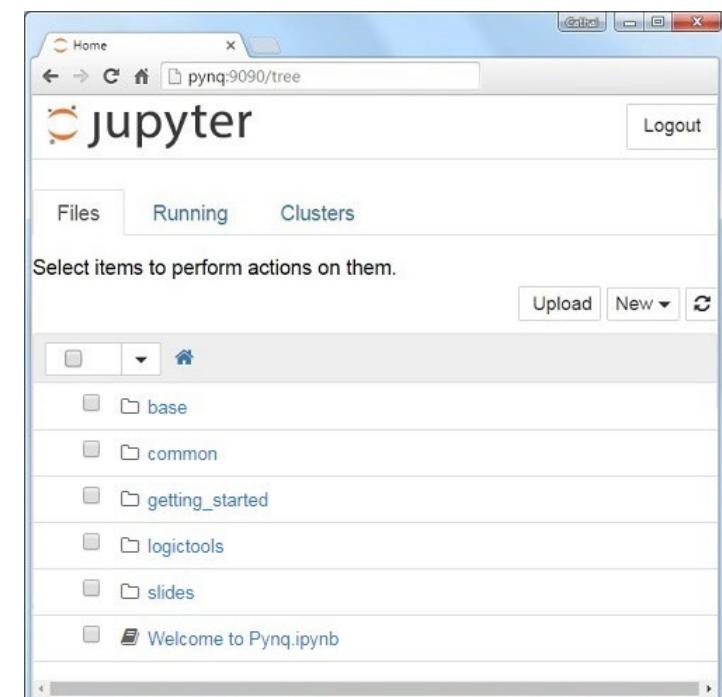
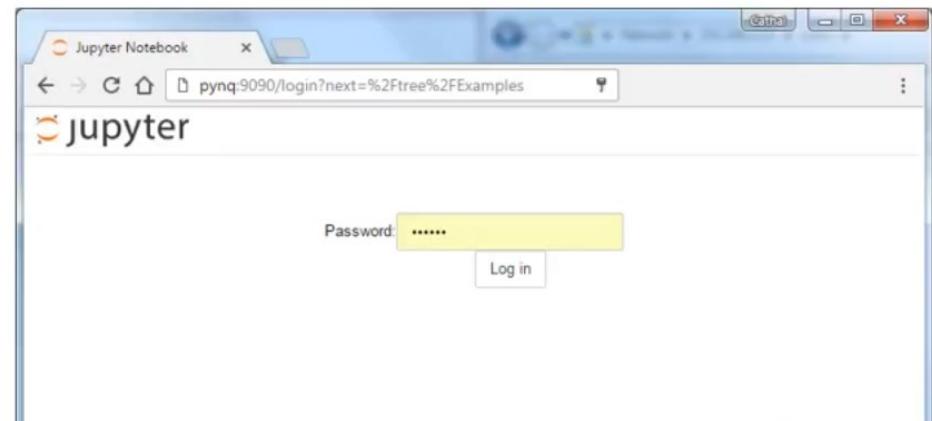
Option B: Connect to a Computer

Getting Started with PYNQ

Connecting to Jupyter portal

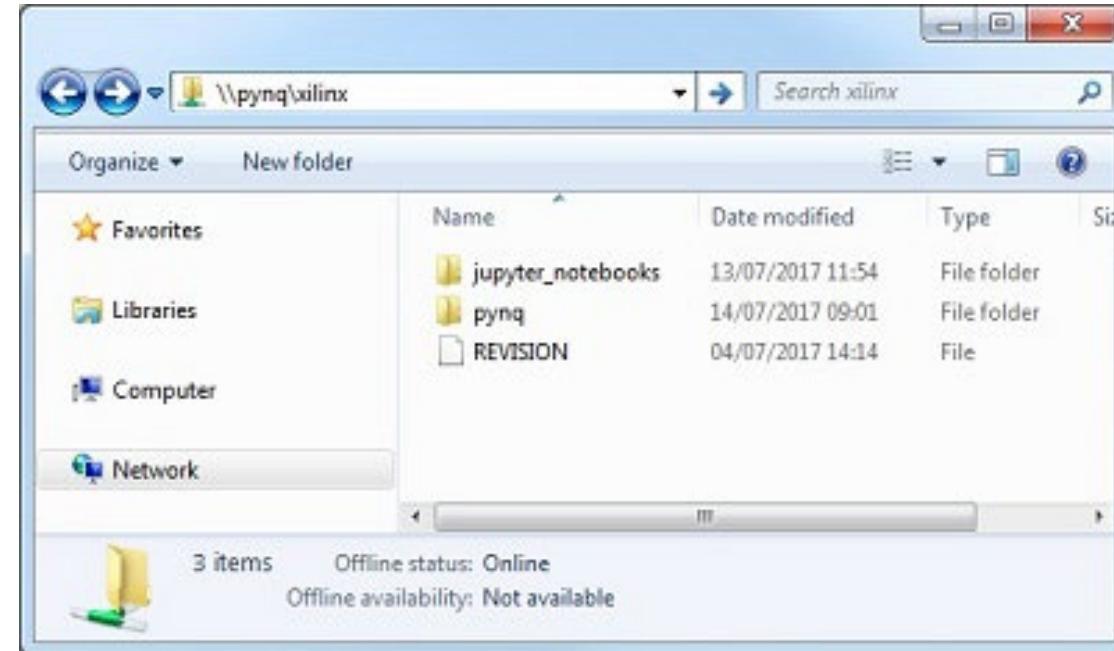
After network connection is established, browse to board's corresponding IP address, then you can start to use the board using Jupyter notebook.

- For boards connected to network:
 - Browse to `http://pynq:9090`
- For boards connected to computer:
 - Browse to <http://192.168.2.99:9090>
- Default password: `xilinx`



Getting Started with PYNQ

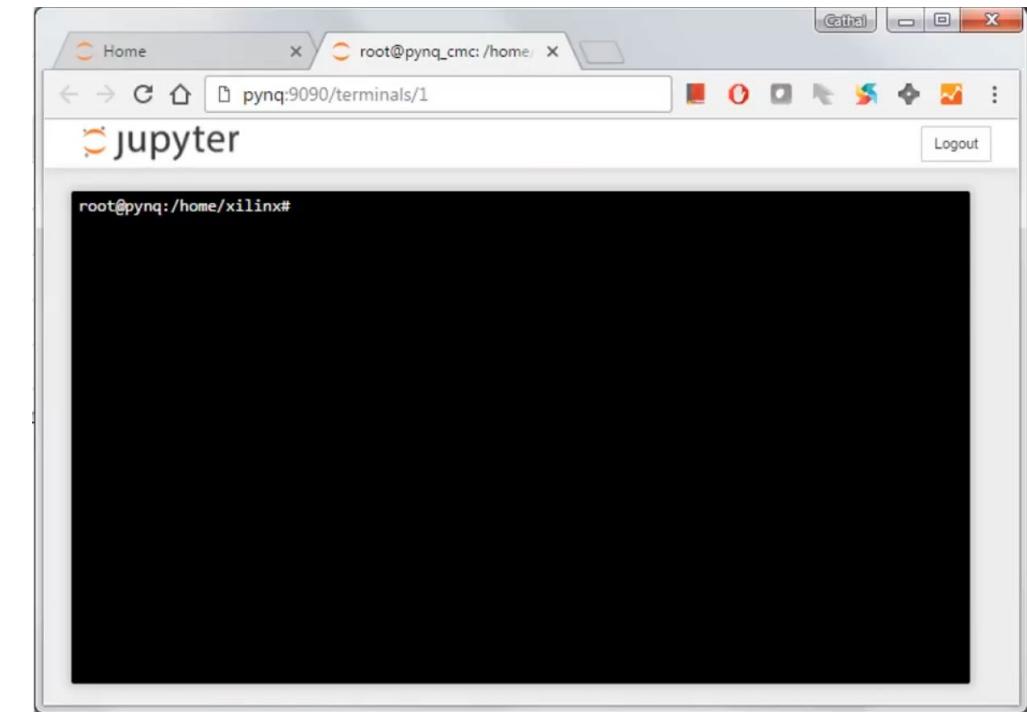
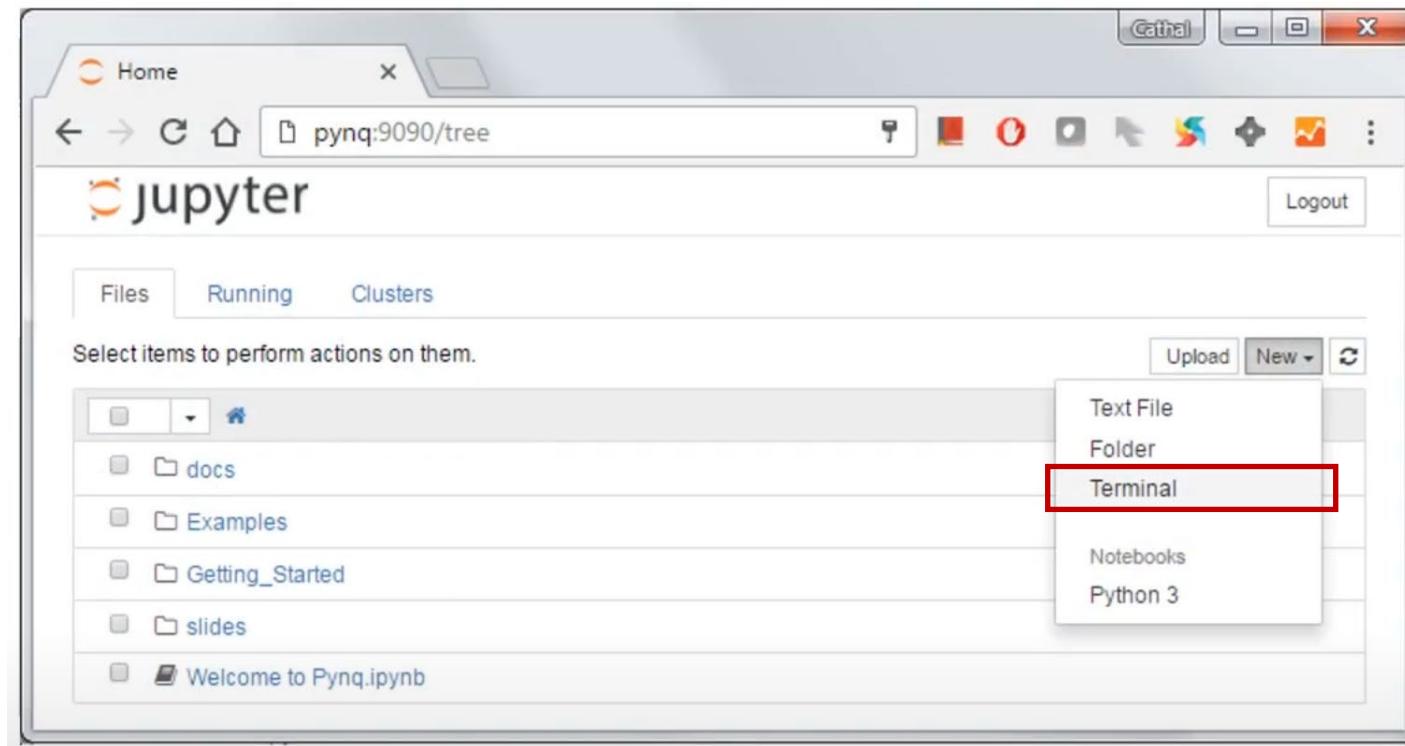
Connect via Samba



- Windows: <\\pynq\xilinx>
- Mac or Linux: <smb://pynq/xilinx>

Getting Started with PYNQ

Jupyter terminal



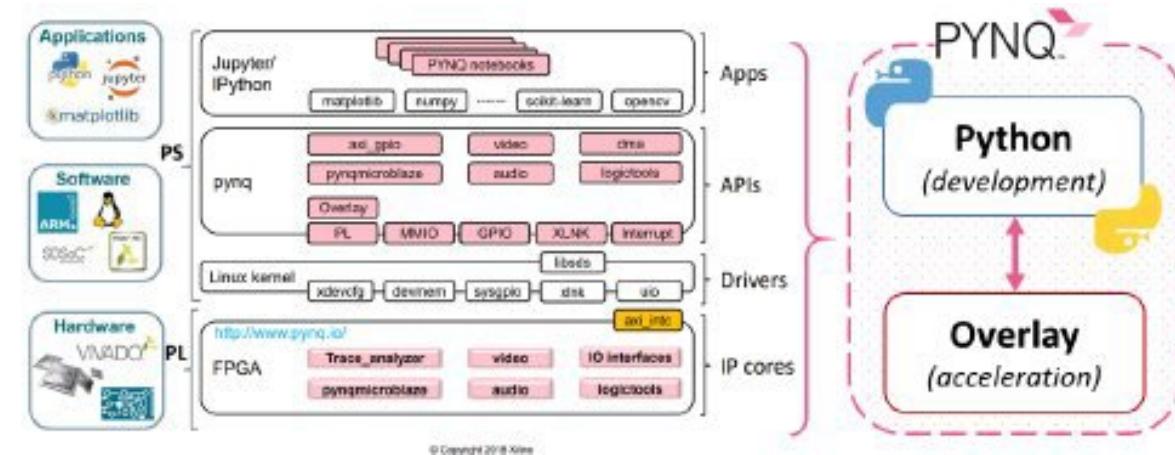
The PS runs a full Linux OS, so you can also *ssh* to the system:

ssh pynq@192.168.2.99

PYNQ Overlays

What are overlays?

- “hardware libraries”, where IP instances become objects
- programmable/configurable FPGA designs
- Can be used in a similar way to a software library to run some functions on the FPGA fabric
- Can be loaded to FPGA dynamically, just like a software library



PYNQ Overlays:

- Provide a Python interface for controlling PL from Python running in the PS
- Created by hardware designers and wrapped with PYNQ Python API
- Each entry (*IPs* or *ports*) in the hardware becomes an **object** and has specific attributes and methods (e.g. LED IO ports can be accessed with `overlay.leds`)

An overlay usually includes:

- A *bitstream* to configure FPGA fabric
- A Vivado design *Tcl file* to determine the available IPs
- Python API that exposes the IPs as attributes (PYNQ library)

Roughly, overlay is:

bitstream + block design structure file + APIs

PYNQ Overlays

Loading an Overlay:

- The PYNQ **Overlay** class can be used to load an overlay

```
from pynq import Overlay
overlay = Overlay("base.bit")
```

- An overlay can be instantiated by specifying the bitstream file
- Overlay instantiation also downloads the bitstream to FPGA

Inspecting an Overlay:

- Once overlay is instantiated, **help()** method can be used to discover what is in an overlay

```
help(overlay)
```

- help()** can also be used to get more information about a specific object in the overlay

```
help(overlay.leds)
```

- An example output from calling **help(base_overlay)**:

Help on BaseOverlay in module pynq.overlays.base object:

```
class BaseOverlay(pynq.overlay.Overlay)
    The Base overlay for the Pynq-Z1

    This overlay is designed to interact with all of the on board peripherals
    and external interfaces of the Pynq-Z1 board. It exposes the following
    attributes:

Attributes
-----
leds : AxiGPIO
    4-bit output GPIO for interacting with the green LEDs LD0-3
buttons : AxiGPIO
    4-bit input GPIO for interacting with the buttons BTN0-3
switches : AxiGPIO
    2-bit input GPIO for interacting with the switches SW0 and SW1
rgbleds : [pynq.board.RGBLED]
    Wrapper for GPIO for LD4 and LD5 multicolour LEDs
video : pynq.lib.video.HDMIWrapper
    HDMI input and output interfaces
audio : pynq.lib.audio.Audio
    Headphone jack and on-board microphone
```

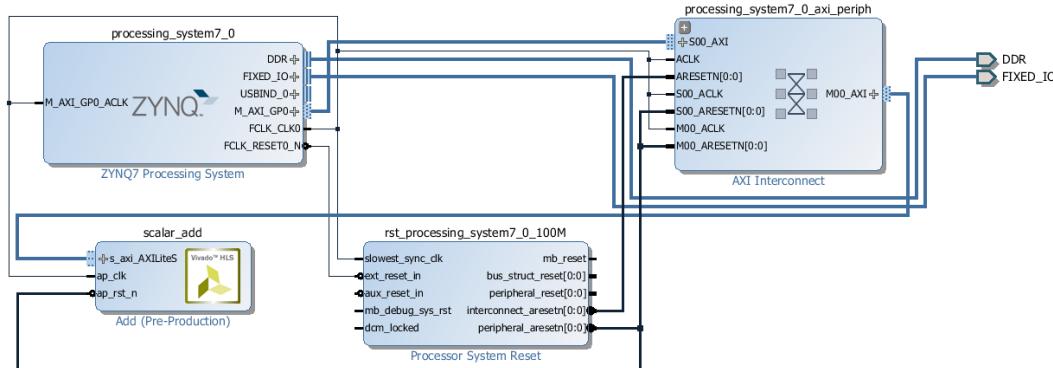
- An API can be used to control the object. For example, turning on LED0 on the board:

```
base_overlay.leds[0].toggle()
```

PYNQ Overlays

Example of Creating and Using Overlay

Assume we create an IP `scalar_add`, and create a block diagram (BD):



This block diagram consists of the IP and glue logic to connect to PS IP.

After synthesis, we get the *bitstream file* (say, `hw.bit`) and the *BD structure file* (`tcl` file or `hwh` file).

With these two files, we can wrap them with PYNQ Overlay class to create a PYNQ overlay (both files should be in the same directory):

```
from pynq import Overlay
overlay = Overlay('hw.bit')
```

Now we get the PYNQ overlay object “`overlay`”.

(note that we get this overlay by bitstream + BD structure + PYNQ API)

Creating the overlay will automatically download the bitstream to FPGA

After creating the overlay, we can inspect the overlay. In Jupyter notebook, we can use a question mark to find out what is inside:

```
overlay?
```

All the entries in the overlay are accessible via attributes on the overlay class. For example, we can access `scalar_add` IP:

```
add_ip = overlay.scalar_add
```

We can also expose the register map associated with IP:

```
add_ip.register_map
```

It prints:

```
RegisterMap {
    a = Register(a=0),
    b = Register(b=0),
    c = Register(c=0),
    c_ctrl = Register(c_ap_vld=1, RESERVED=0)
}
```

We can also interact with the IP using the register map:

```
add_ip.register_map.a = 3
add_ip.register_map.b = 4
Print(add_ip.register_map.c)
```

Alternatively, by reading the driver source code generated by HLS we can determine the offsets we need to write the two arguments (they are all in the same memory space):

```
add_ip.write(0x10, 4)
add_ip.write(0x18, 5)
add_ip.read(0x20)
```

PYNQ Overlays – Prepare Input/Output Buffers

Allocate

- The `pynq.allocate` function is used to allocate memory that will be used by IP in the PL
- `pynq.allocate` function returns a `pynq.Buffer` object that is a sub-class of NumPy's `ndarray` with additional properties and methods suited for use with the programmable logic
 - `device_address` is the address that should be passed to the programmable logic to access the buffer
 - `coherent` is True if the buffer is cache-coherent between the PS and PL
 - `flush` flushes a non-coherent or mirrored buffer ensuring that any changes by the PS are visible to the PL
 - `invalidate` invalidates a non-coherent or mirrored buffer ensuring any changes by the PL are visible to the PS
 - `sync_to_device` is an alias to `flush`
 - `sync_from_device` is an alias to `invalidate`

Example: Create a contiguous array of 5 32-bit unsigned integers

```
from pynq import allocate
input_buffer = allocate(shape=(5,), dtype='u4')
input_buffer[:] = range(5)
input_buffer.flush()
```

PYNQ Overlays – Running Accelerators

After creating the overlay and have data buffers ready, we can start the accelerator.

Running Accelerators

Start the kernel synchronously:

```
ol.my_kernel.call(input_buf, output_buf)
```

The call function has the same function signature as the top function in original HLS source code.

Alternatively, start the kernel in a non-blocking way:

```
handle = ol.my_kernel.start(input_buf, output_buf)  
handle.wait()
```

Freeing designs (overlays):

```
ol.free()
```

Execution results can be collected by accessing output buffers.

NOTE: call, start, and wait are newly added since PYNQ 2.5.

For older version of PYNQ, to invoke the accelerator, we need to manually set the *ap_start* bit of the IP, and then wait for the *ap_start* signal.

The address of these bits can be found from the driver file generated by Vivado HLS.

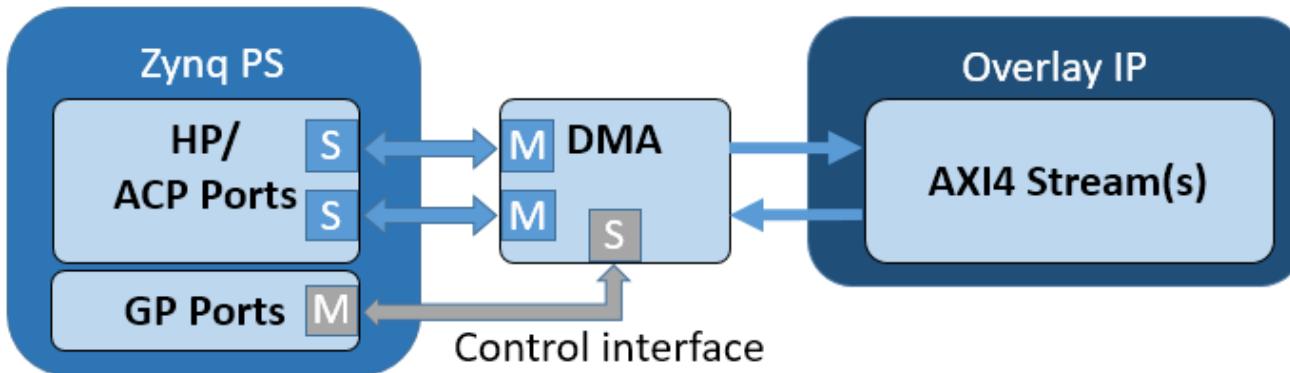
PYNQ Libraries

PYNQ provides a Python API for common peripherals and PL control

- Audio/Video
- GPIO devices (buttons, switches, LEDs, etc.)
- Headers and IO pins (e.g. Raspberry Pi header)
- PynqMicroBlaze subsystem
- Low-level PL control e.g. memory-mapped IO, memory allocation, overlay control, etc.

PYNQ Libraries - DMA

- DMA (direct memory access) can be used for high performance burst transfers between PS DRAM and the PL.
- PYNQ supports the AXI central DMA IP with the PYNQ DMA class.



```
overlay = Overlay('example.bit')
dma = overlay.axi_dma
# allocate arrays
input_buffer = allocate(shape=(5,), dtype=np.uint32)
output_buffer = allocate(shape=(5,), dtype=np.uint32)
# write some data to input array
for i in range(5):
    input_buffer[i] = i
```

```
# actual compute
...
# transfer data using DMA
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```

PYNQ Libraries – PYNQ MicroBlaze Subsystem

The PYNQ MicroBlaze subsystem allows loading of programs from Python, controlling executing by triggering the processor reset signal, reading and writing to shared data memory, and managing interrupts received from the subsystem.

- Each PYNQ MicroBlaze subsystem is contained within an IO Processor (IOP)
- An IOP defines a set of communication and behavioral controllers that are controlled by Python.
- Supported IOPs: Arduino, Grove, Pmod, Raspberry Pi

Example:

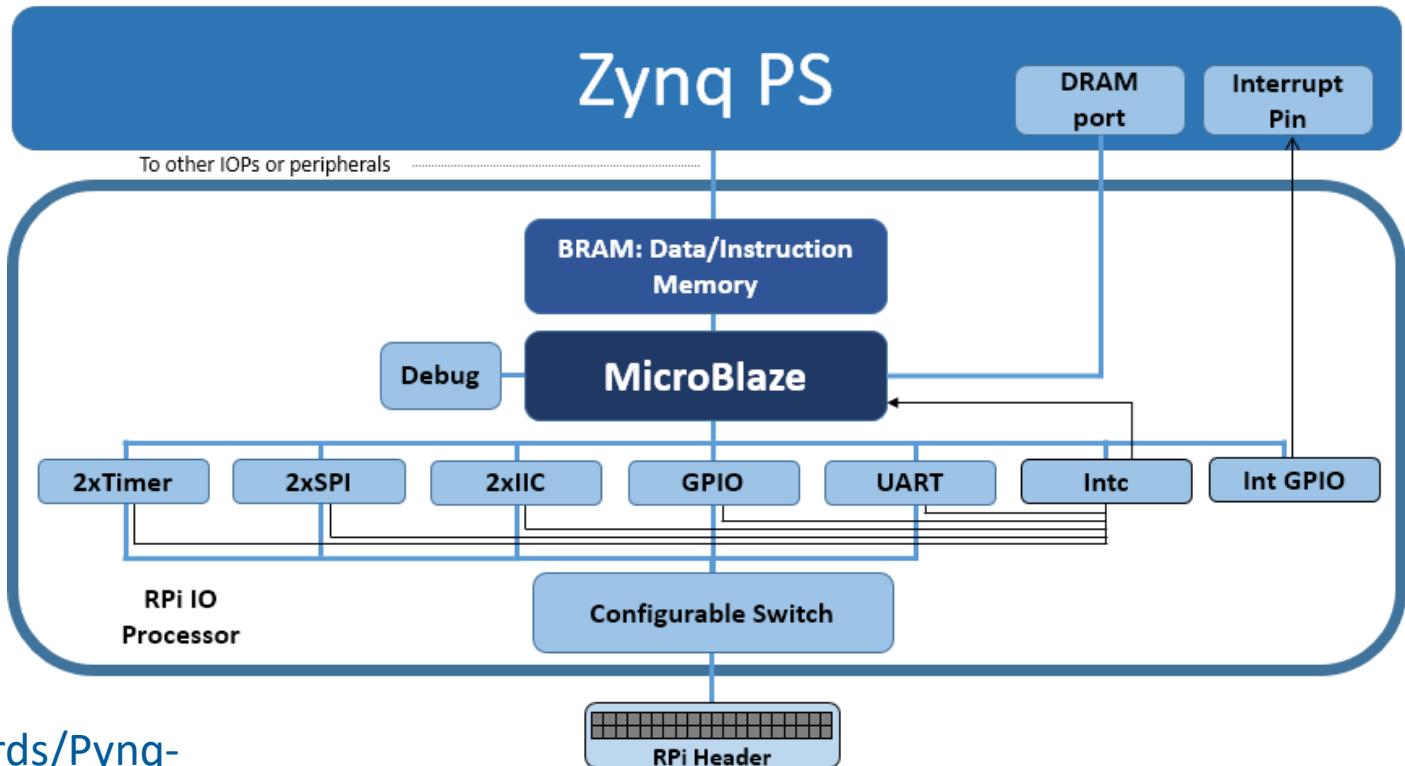
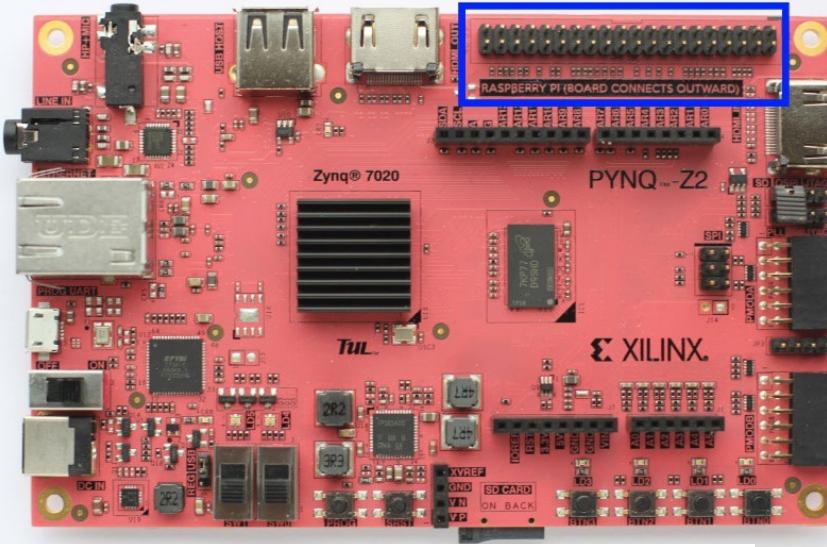
```
from pynq.overlay import BaseOverlay
from pynq.lib import PynqMicroblaze

base = BaseOverlay('base.bit')

mb = PynqMicroblaze(base.iop1.mb_info, # iop1 corresponds to Pmod A connector
                     "/home/xilinx/pynq/lib/pmod/pmod_timer.bin")
mb.reset()
```

PYNQ Libraries – Raspberry Pi Header

- The rpi subpackage is a collection of drivers for controlling peripherals attached to a RPi (Raspberry Pi) interface.
- The RPi PYNQ MicroBlaze is available to control the RPi interface
- RPi PYNQ MicroBlaze has a PYNQ MicroBlaze Subsystem, a configurable switch, and the following AXI controllers:
 - 2x AXI I2C
 - 2x AXI SPI
 - 1x AXI GPIO
 - 2x AXI Timer
 - 1x AXI UART
 - AXI Interrupt controller
 - Interrupt GPIO
 - Configurable Switch



Example: Reading Values from Touch Keypad:

https://github.com/Xilinx/PYNQ/blob/master/boards/Pynq-Z2/base/notebooks/rpi/rpi_touchpad.ipynb

PYNQ on XRT Platforms

Besides Xilinx Zynq platforms, PYNQ also support XRT-based platforms such as Amazon's AWS F1 and Alveo for cloud and on-premise deployment.

(XRT: Xilinx Runtime Library)

Running Accelerators

Start the kernel synchronously:

```
ol.my_kernel.call(input_buf, output_buf)
```

The call function has the same function signature as the top function in original HLS source code.

Alternatively, start the kernel in a non-blocking way:

```
handle = ol.my_kernel.start(input_buf, output_buf)  
handle.wait()
```

Freeing designs (overlays):

```
ol.free()
```

Efficient Scheduling of Multiple Kernels

start and **call** have an optional keyword parameter **waitfor** that can be used to create a dependency graph which is executed in the hardware.

```
handle = ol.vadd_1.start(input1, input2, output)  
ol.vadd_1.call(input3, output, output, waitfor=(handle,))
```

Multiple FPGA Cards

PYNQ supports multiple accelerator cards in one server. It provides a **Device** class to designate which card should be used for given operations.

```
> for i in range(len(pynq.Device.devices)):  
>   print("{} {}".format(i, pynq.Device.devices[i].name))  
0) xilinx_u200_xdma_201830_2  
1) xilinx_u250_xdma_201830_2  
2) xilinx_u250_xdma_201830_2  
3) xilinx_u250_xdma_201830_2
```

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 5:

High-Level Synthesis

Sitao Huang

sitaoh@uci.edu

February 1, 2022

Slide courtesy of Prof. Deming Chen, UIUC, ECE 527: SoC Design



Logistics

- **Homework 1** released, due: ***January 31***, 11:59 PM on Canvas
- Homework 2 expected to release later this week, due February 7
- **Midterm: February 10** (Thursday), 8:00-9:20 AM (in class)
- Midterm review session / Q&A: February 8 (Tuesday)
- Project proposal due: February 14
 - Options: (a) literature review paper or (b) compiler + accelerator project
- In-person (***hybrid***) instruction starting next week (week 5)! Classroom: **SSTR 101**
- First in-person class: ***February 1***

High-Level Synthesis

- **High-level synthesis (HLS)**, also referred to as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavior synthesis
- HLS takes an abstract behavioral specification of a digital system, and finds a register-transfer level structure that realizes the given behavior

INPUT:

- A high-level, algorithmic description
 - Control structures (if/else, loop, subroutines)
 - Concurrent and sequential semantics
 - Abstract data types
 - Logical and arithmetic operators
- A set of constraints
 - Speed, power, area, interconnect style
 - A library of pre-specified components

OUTPUT:

- A register-transfer level description for further synthesis and optimization

High-Level Synthesis

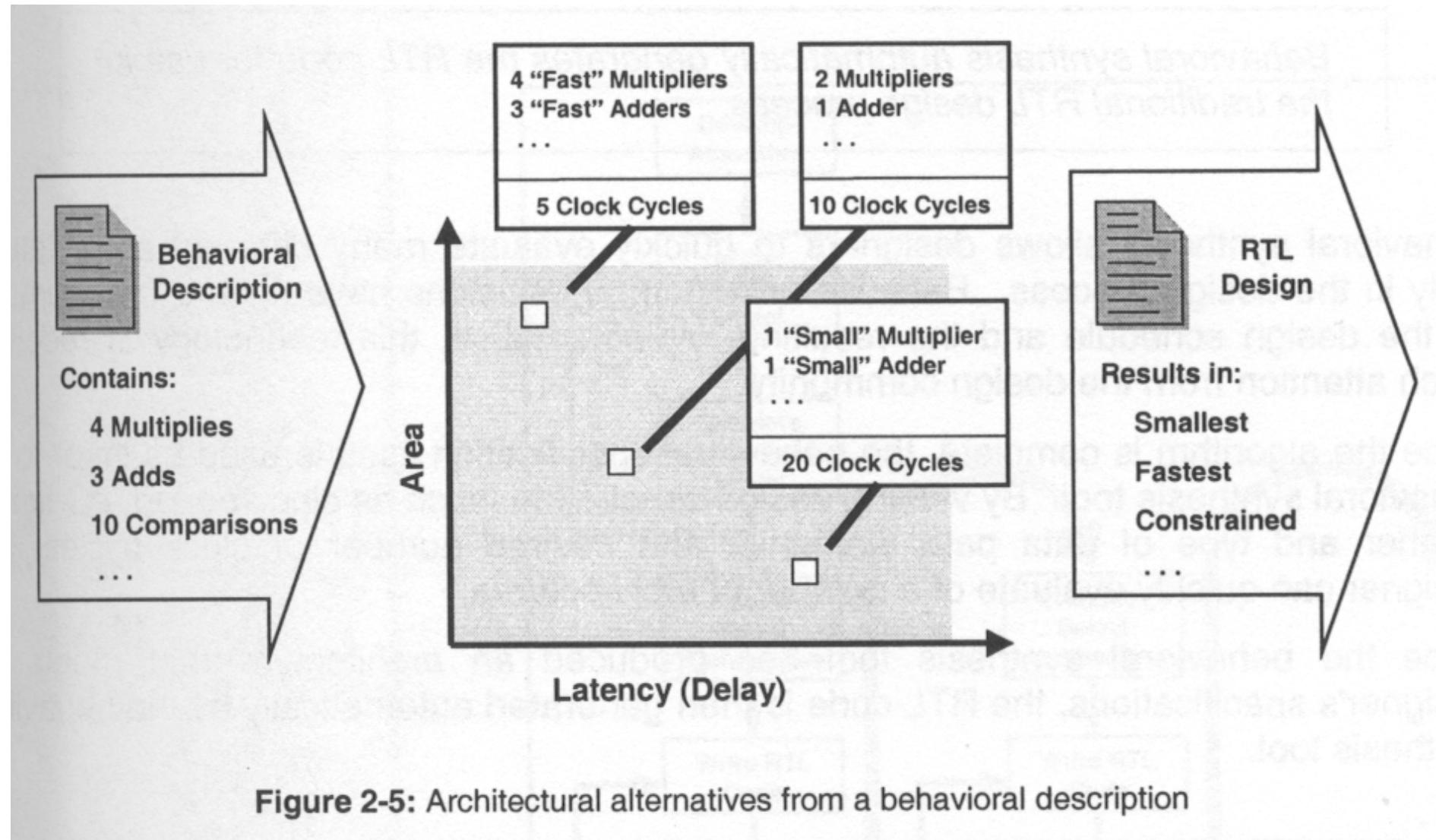
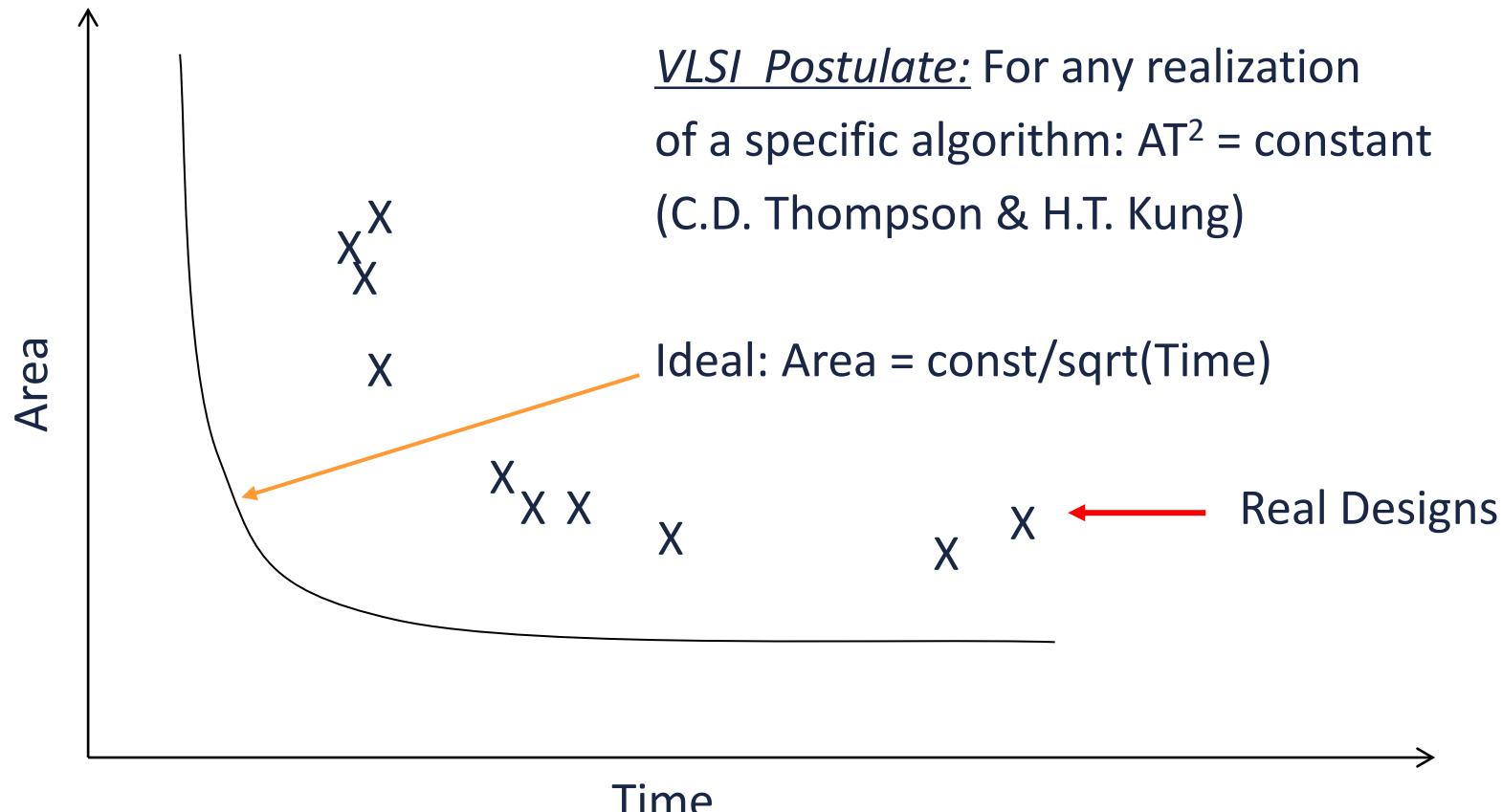


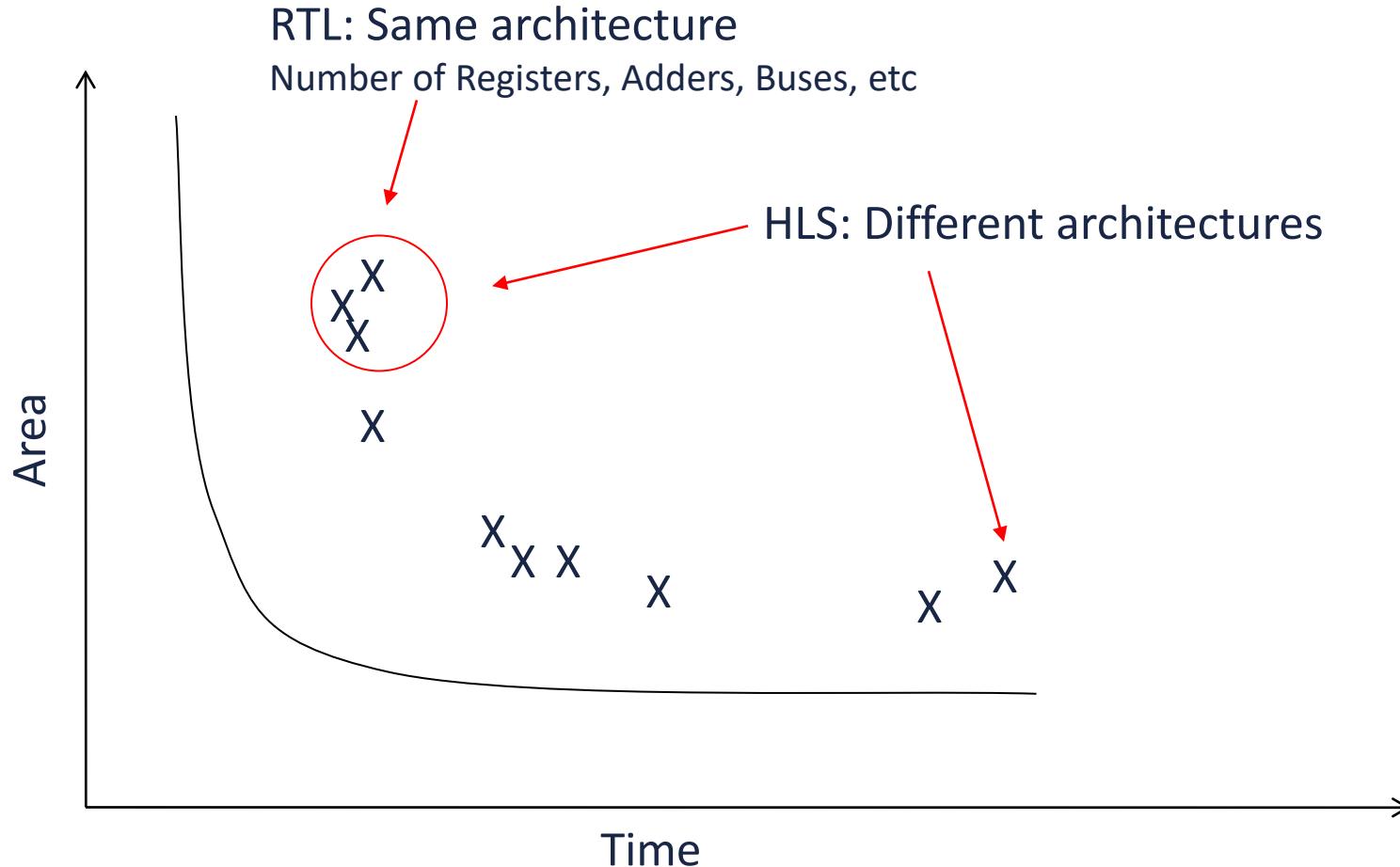
Figure 2-5: Architectural alternatives from a behavioral description

Design Space Exploration



Add the Power tradeoff and this becomes a 3-dimensional graph

Design Space Exploration: RTL vs. Behavioral Synthesis



High-Level Synthesis Process

Resource Allocation:

- Allocating resources (library components) to each of the operations, buses, muxes, and registers for storage

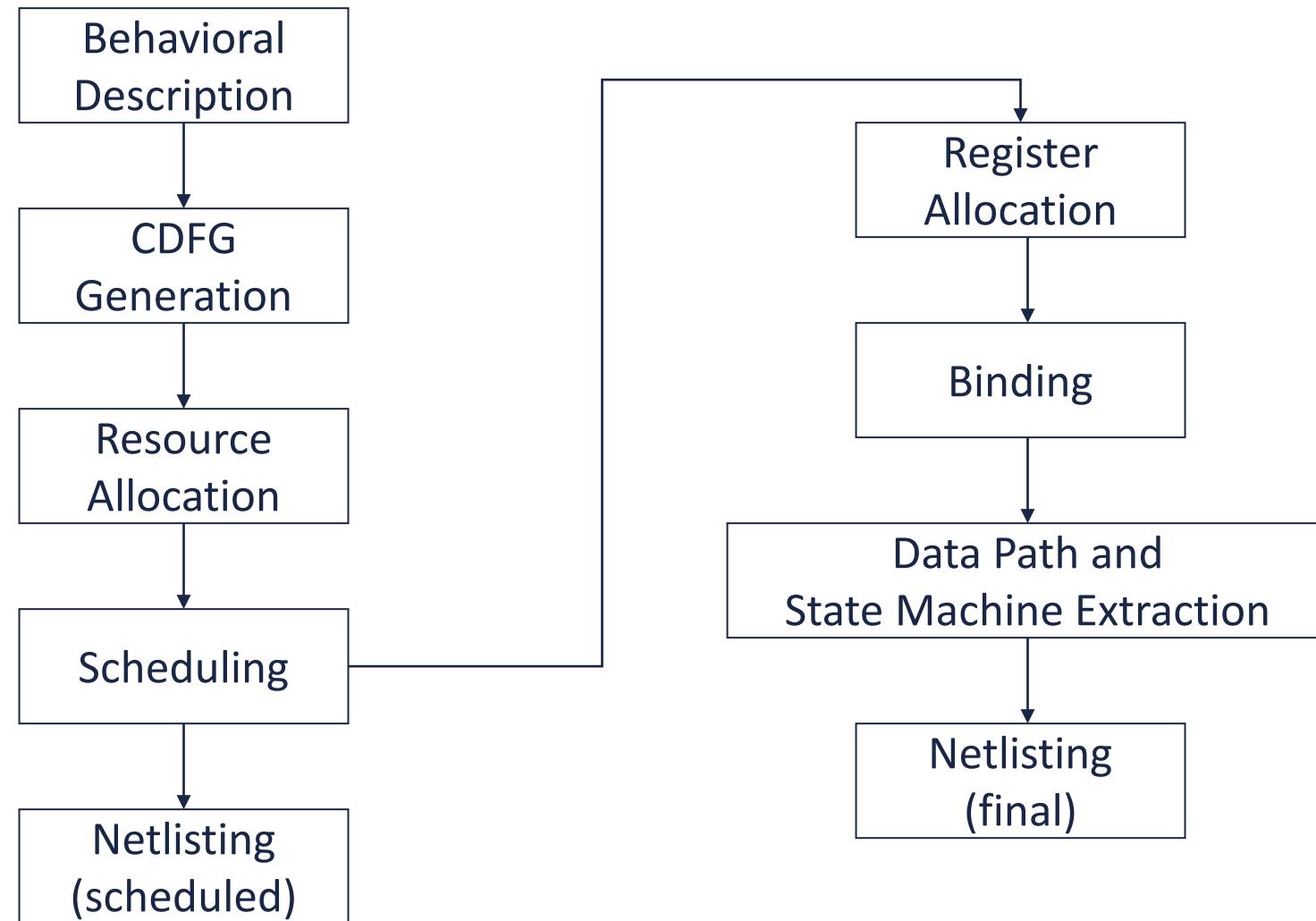
Scheduling:

- Scheduling the operations in the CDFG to minimize area, time and/or power

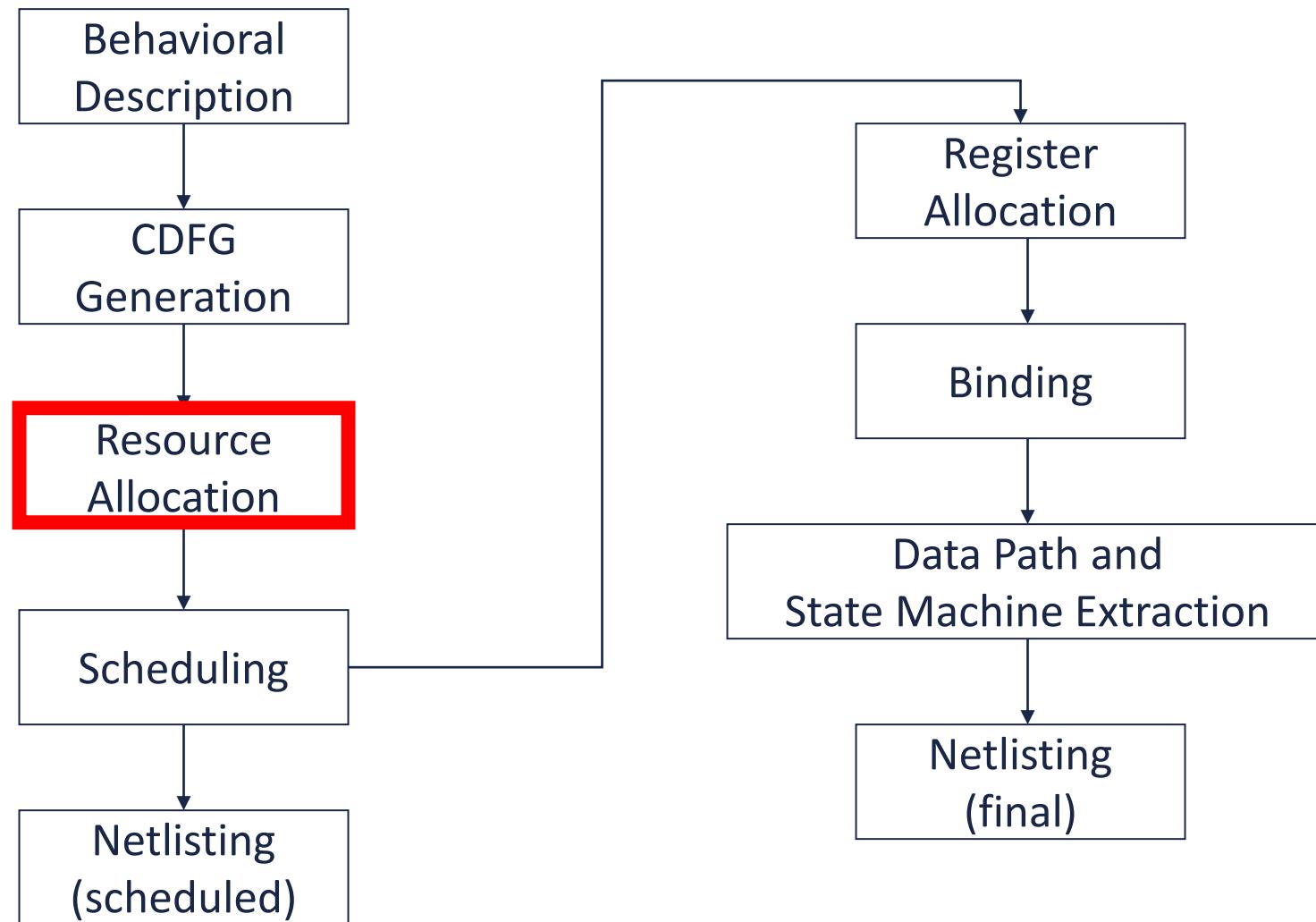
Binding:

- Determining the time of use of each component
- e.g., which registers to use and when

High-Level Synthesis Steps



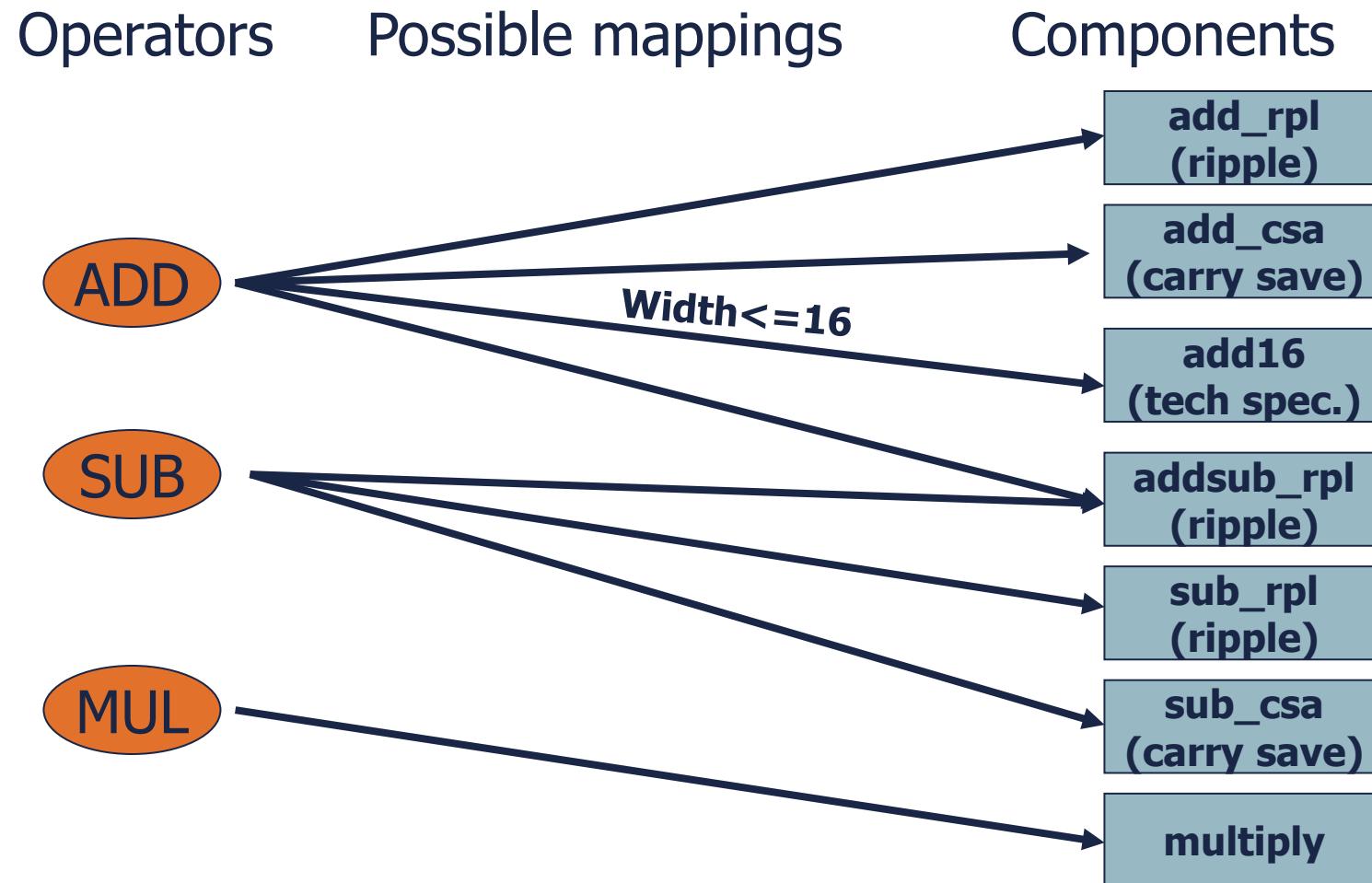
High-Level Synthesis Steps



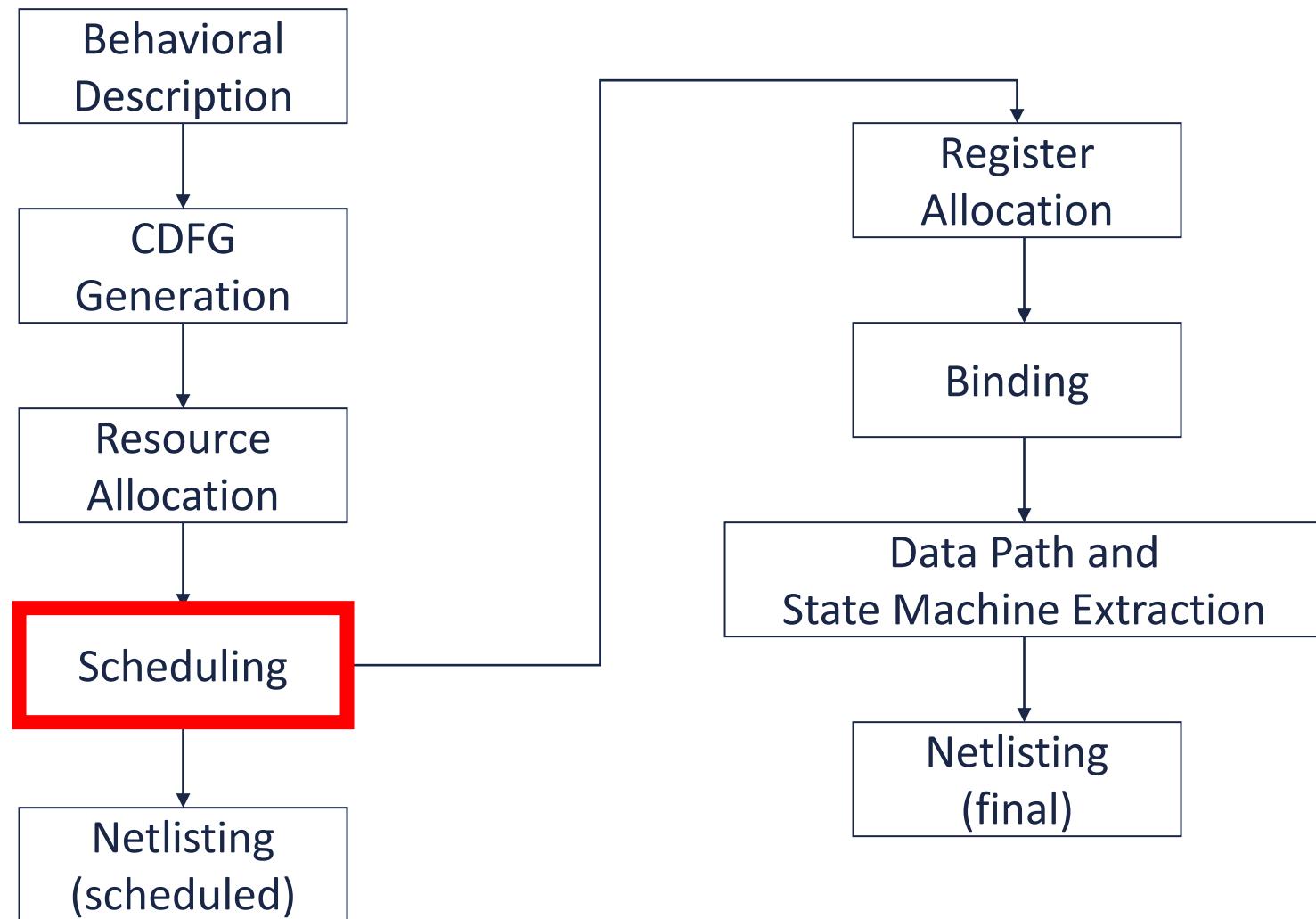
HLS Step: Resource Allocation

- Deciding how many and which kinds of resources will be used in a given implementation
- This has a major impact on final design
 - Number of operation units (multiple adders?) set the maximum parallelism that the architecture can provide
 - Reuse of overloaded operators (e.g., an adder/subtractor unit) provides smallest designs
 - Choice of buses or muxes provides parallelism vs. size
 - Choice of registers, multi-ported register files or RAM also limits parallelism in data movement

Example: Mapping of Operators to Components



High-Level Synthesis Steps



HLS Step: Scheduling

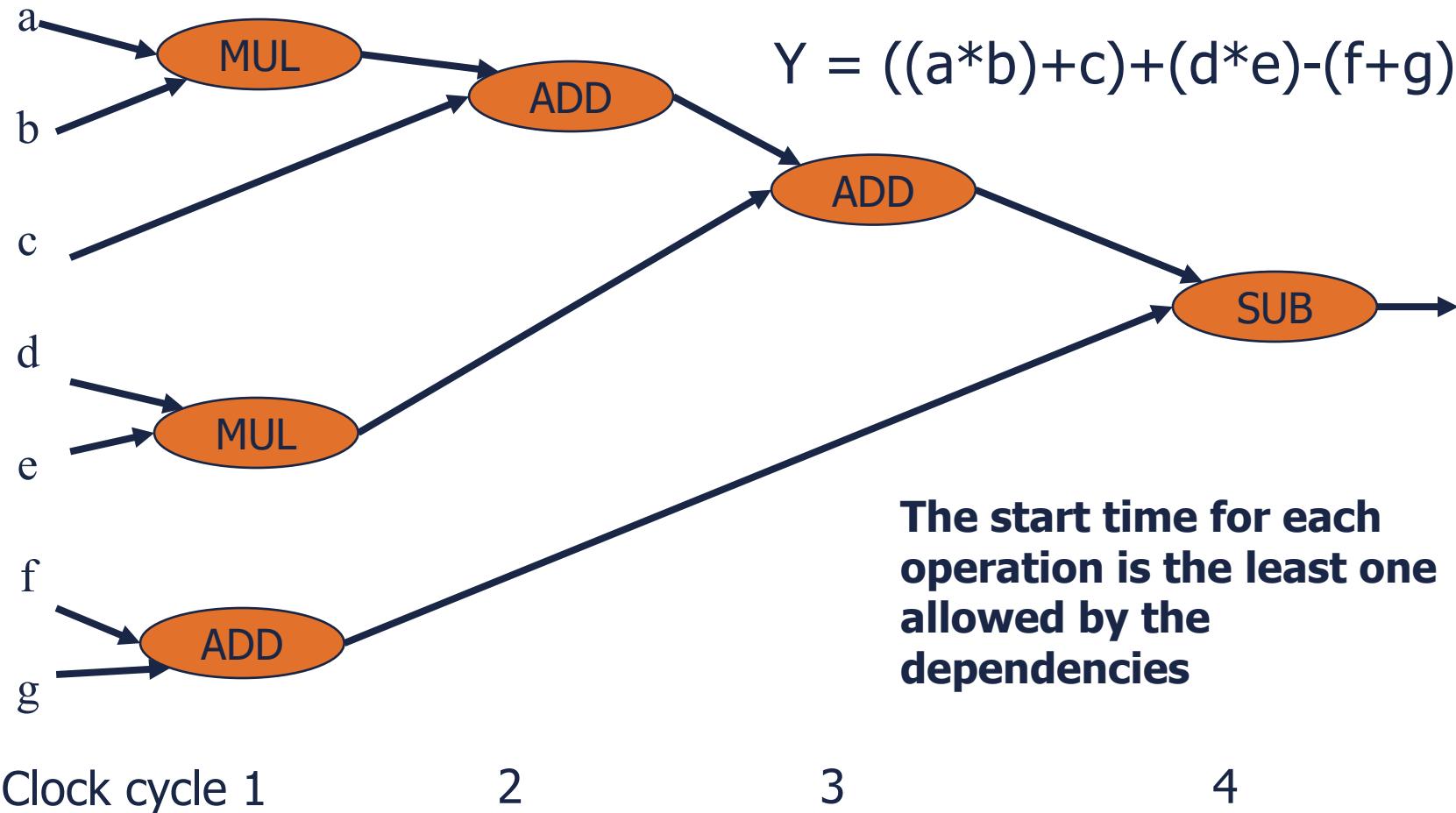
Scheduling:

- Mapping of operations to time slots (cycles)

Basic Algorithms:

- ASAP: As Soon As Possible
- ALAP: As Late As Possible
- List Scheduling

ASAP Schedule (unconstrained)

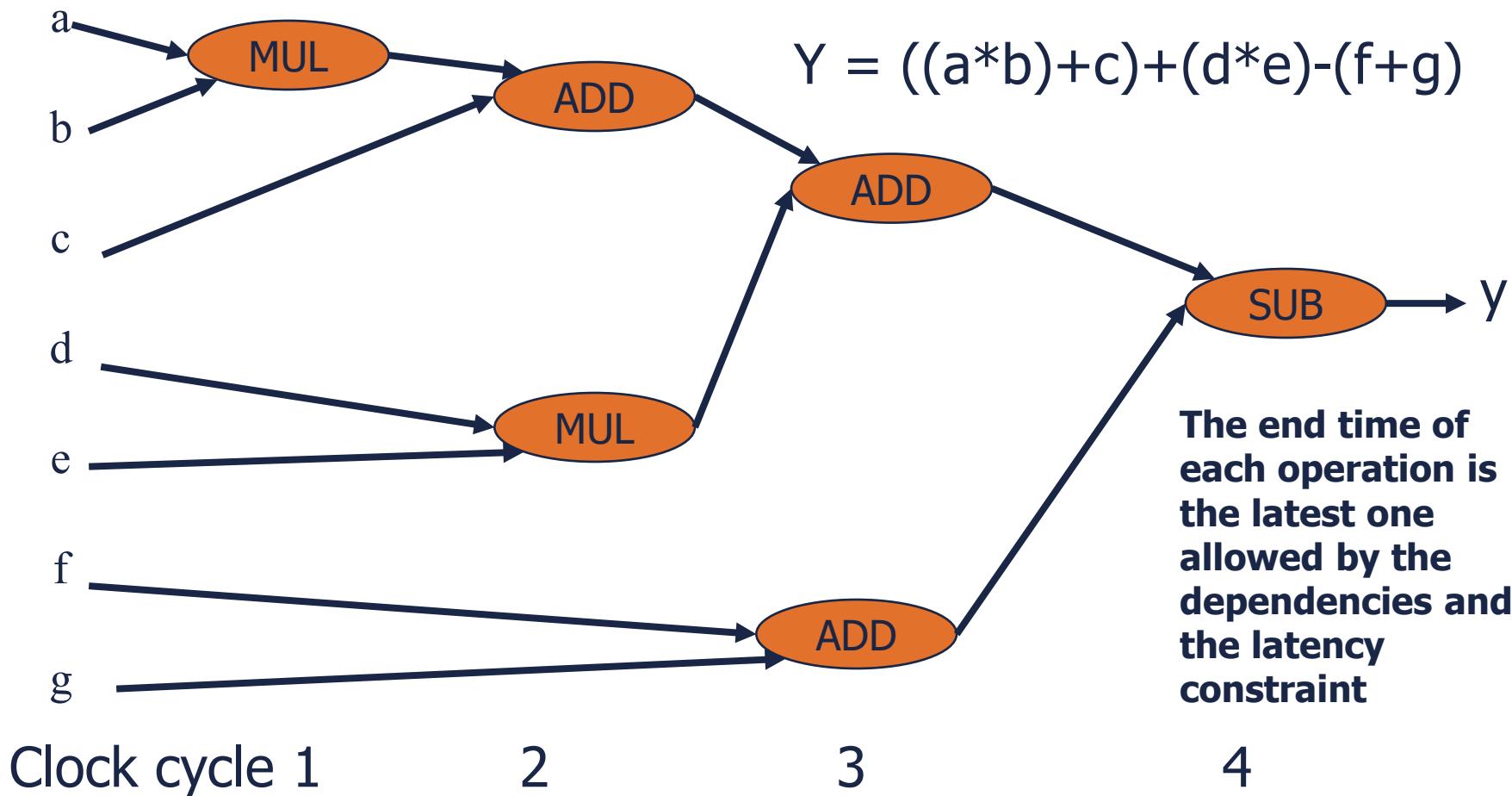


Can we do better than this in terms of the DFG itself? What is the resource usage?

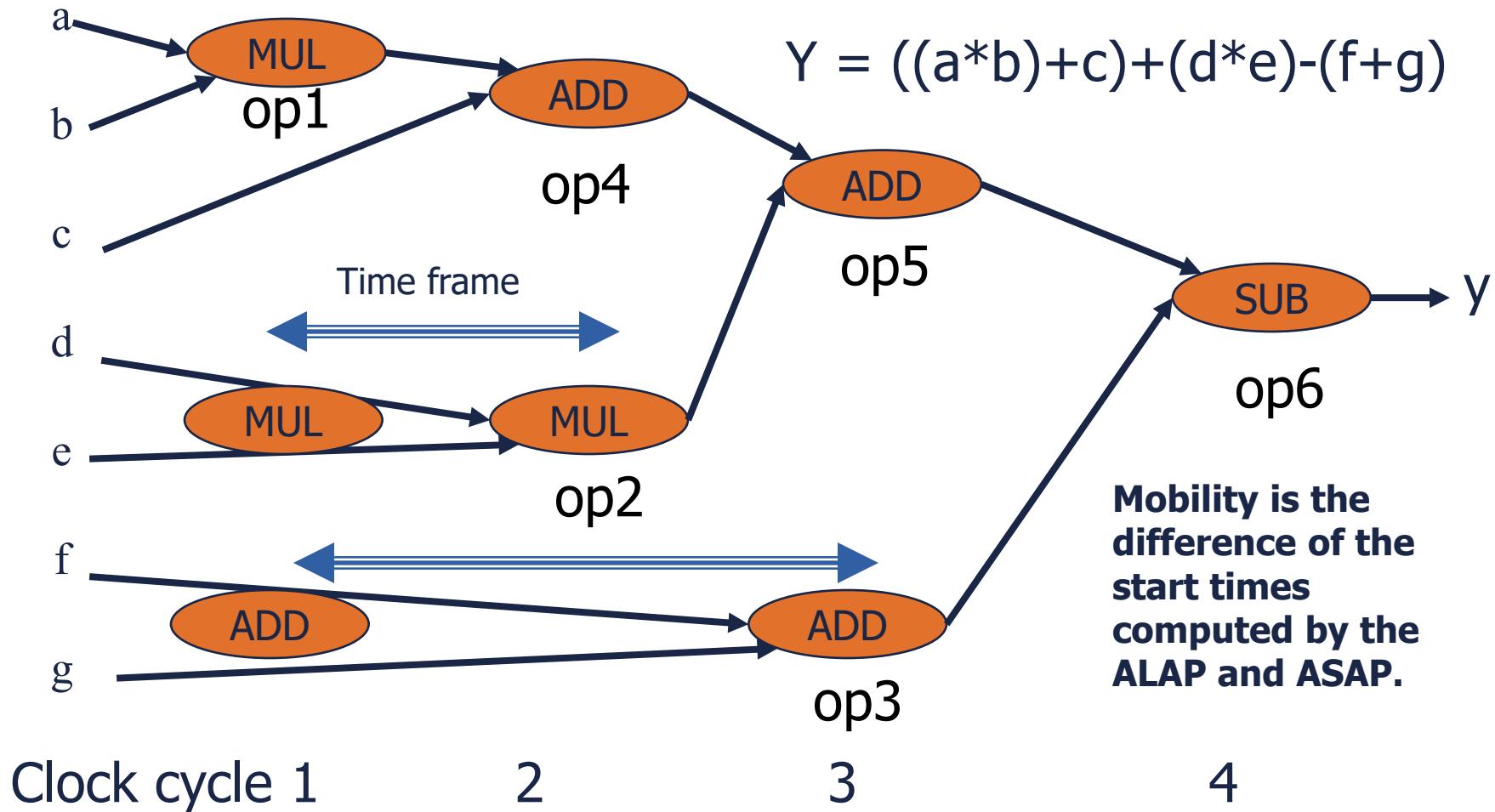
ASAP Algorithm

```
ASAP (G(V, E)) {  
    Schedule all the nodes driven only by PIs to cycle 1,  
    for all such  $v_i$  nodes,  $t_i$  (staring time) = 1;  
    Repeat {  
        Select a vertex  $v_i$  whose predecessors are all scheduled;  
        Schedule  $v_i$  by setting  $t_i = \text{MAX}(t_j + d_j); (v_j, v_i) \in E$   
    }  
    Until all the nodes are scheduled;  
    Return the schedule in a vector;  
}
```

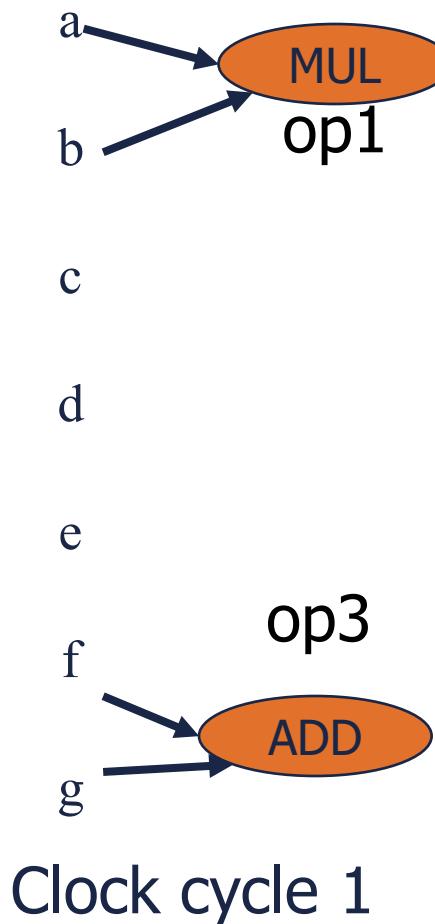
ALAP Schedule



Mobility (or Slack)



List Scheduling (1)



Schedule
op1 and op3

Prioritized Ready List
op1(mul)0
op2(mul)1
op3(add)0

- Priority based on mobility (other metrics possible)
- Resource constraints: one adder, one multiplier
- Schedule ready nodes

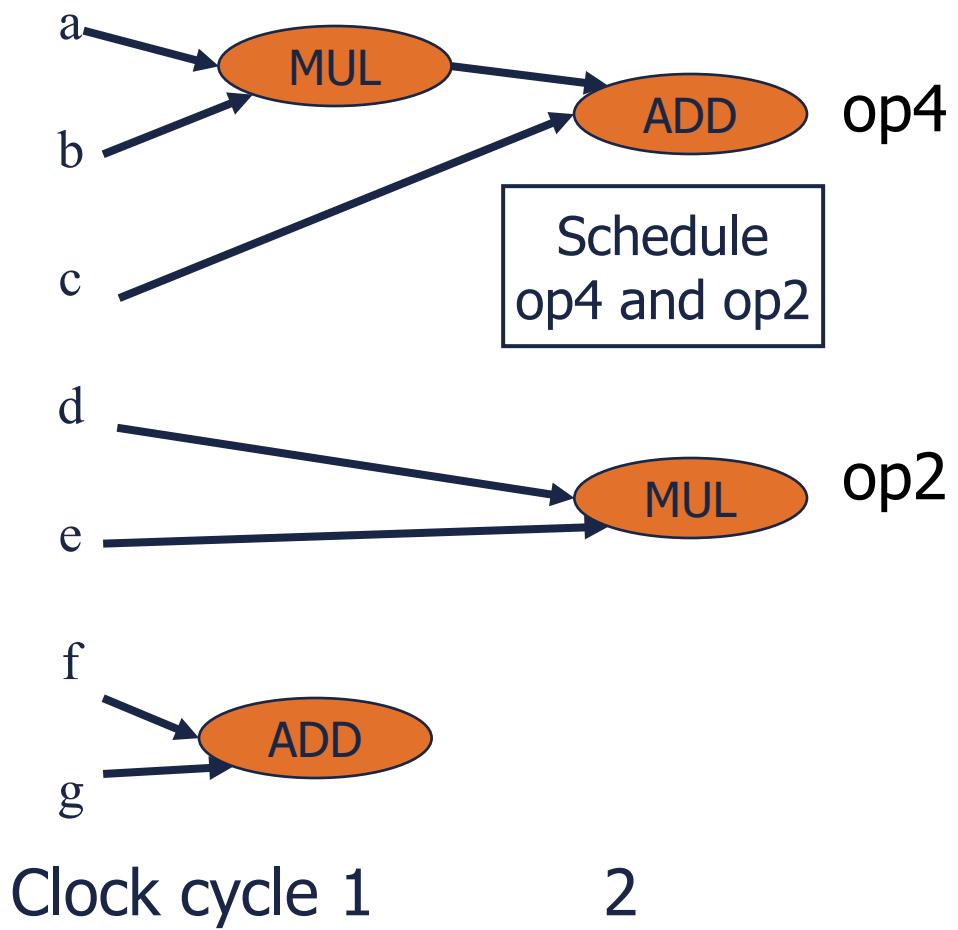
Clock cycle 1

2

3

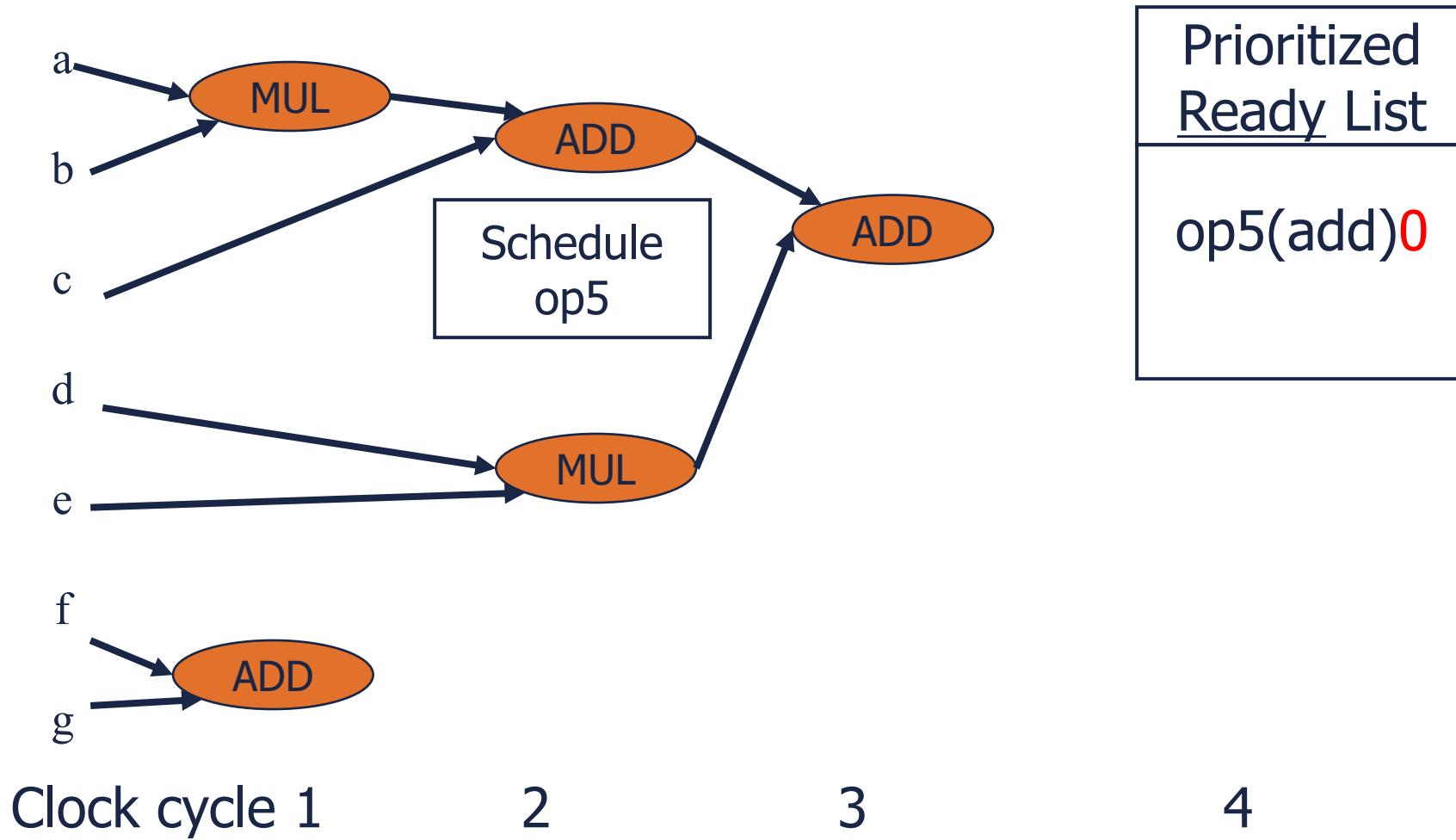
4

List Scheduling (2)

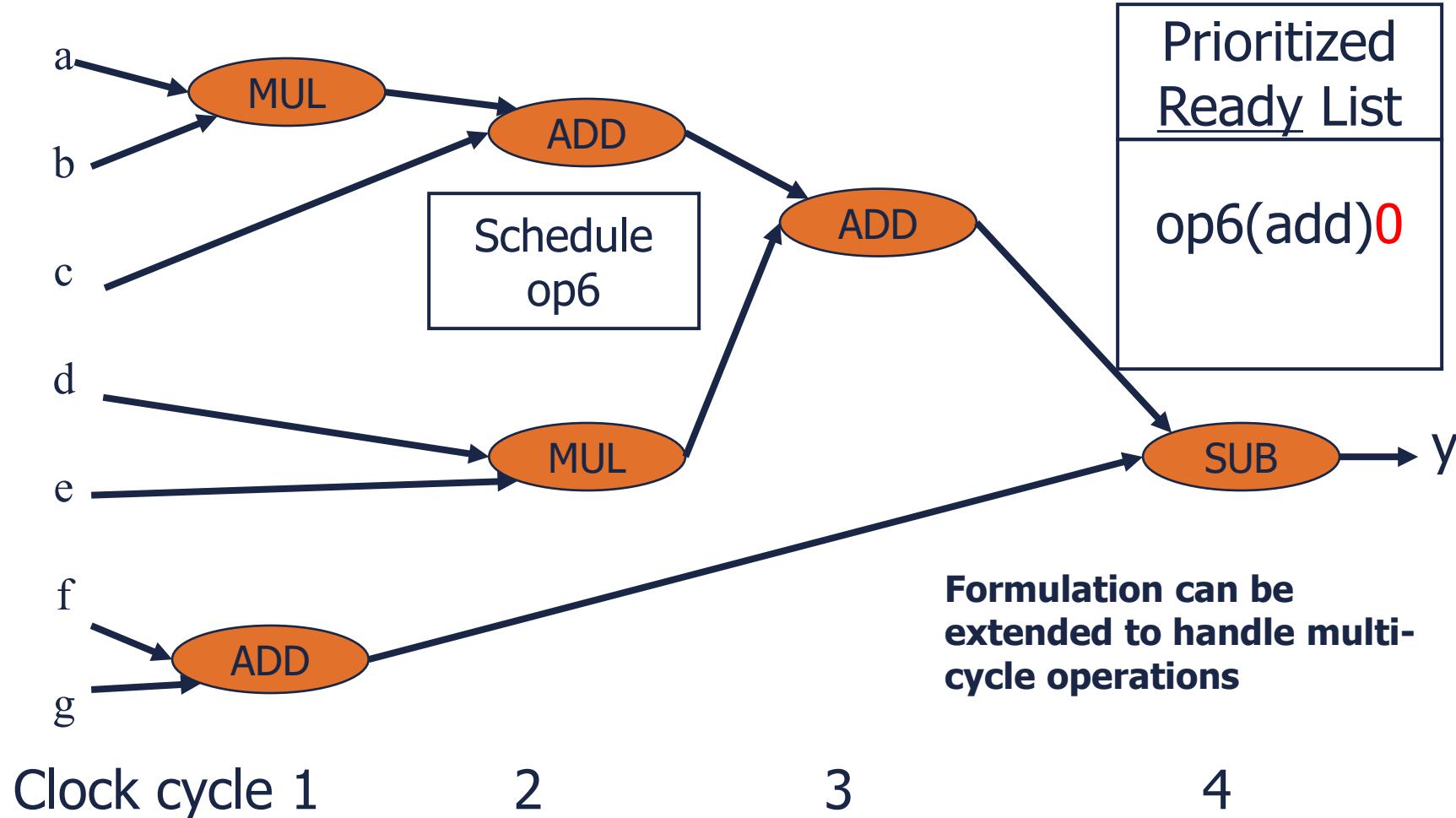


Prioritized Ready List
op2(mul)0
op4(add)0

List Scheduling (3)



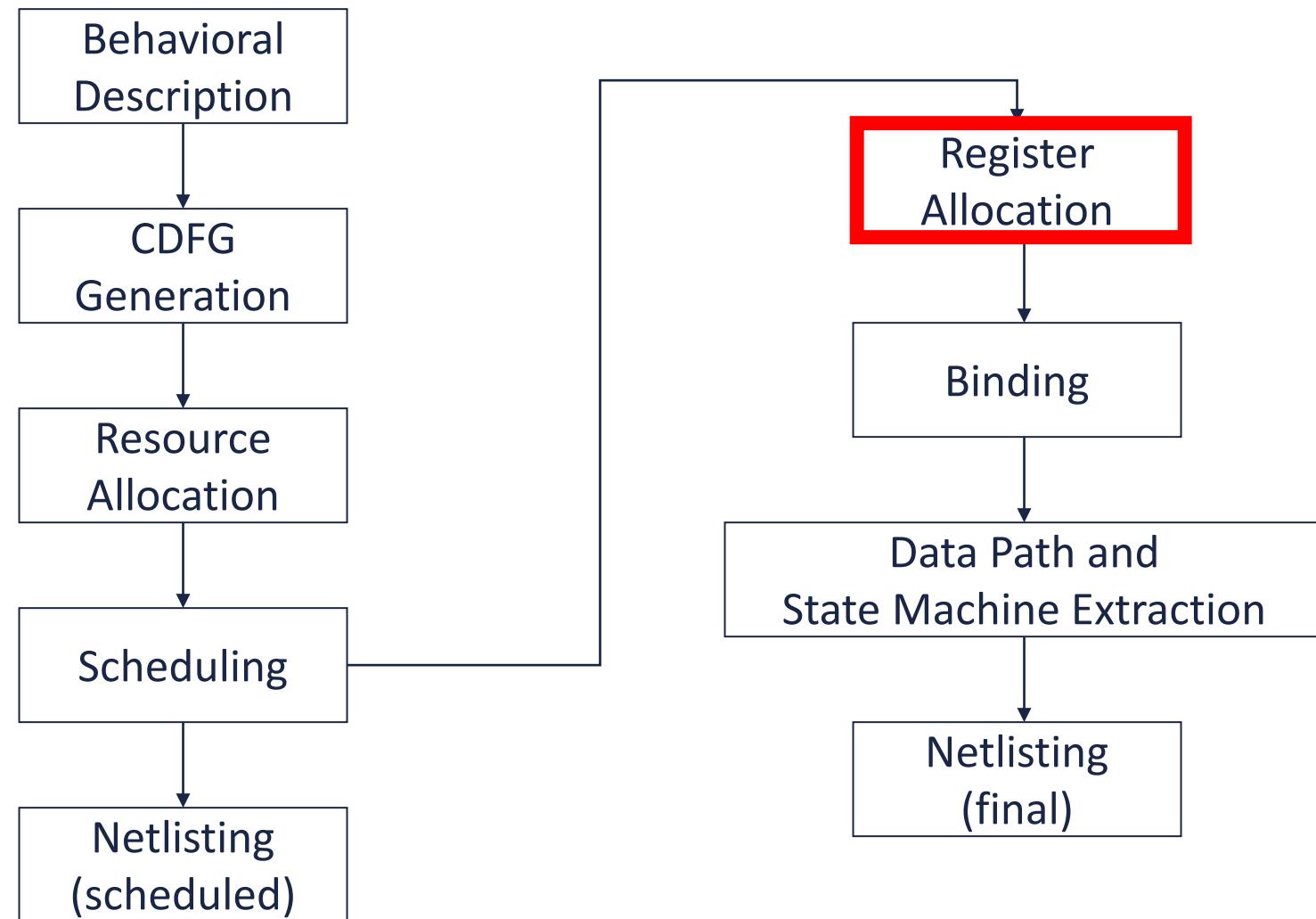
List Scheduling (4)



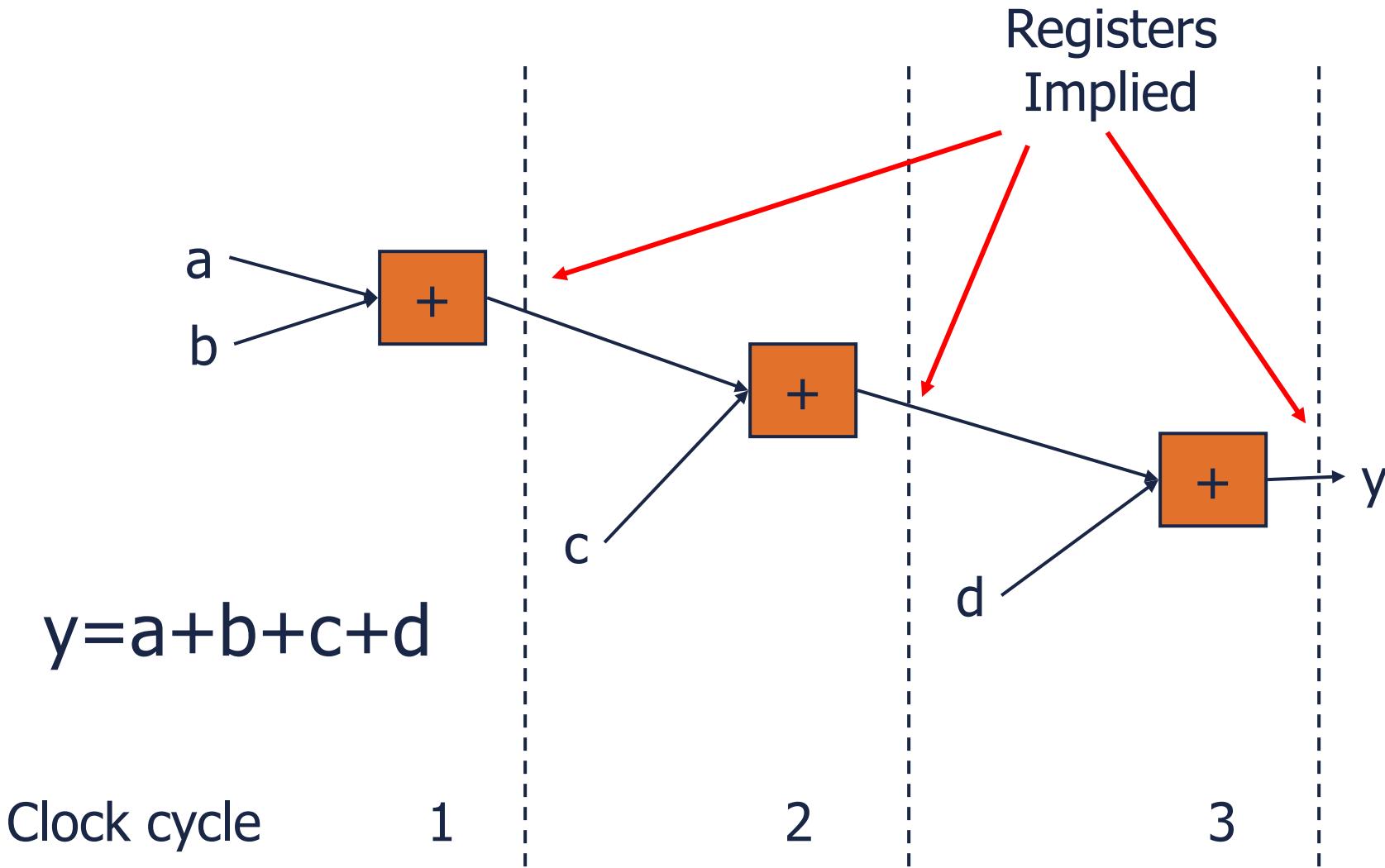
List Scheduling Algorithm

```
LIST_L( $G(V, E)$ ,  $a$ ) {
     $l = 1$ ;
    repeat {
        for each resource type  $k = 1, 2, \dots, n_{res}$  {
            Determine candidate operations  $U_{l,k}$ ;
            Determine unfinished operations  $T_{l,k}$ ;
            Select  $S_k \subseteq U_{l,k}$  vertices, such that  $|S_k| + |T_{l,k}| \leq a_k$ ;
            Schedule the  $S_k$  operations at step  $l$  by setting
                 $t_i = l$  for all  $i : v_i \in S_k$ ;
        }
         $l = l + 1$ ;
    }
    until (all nodes are scheduled);
    return ( $t$ );
}
```

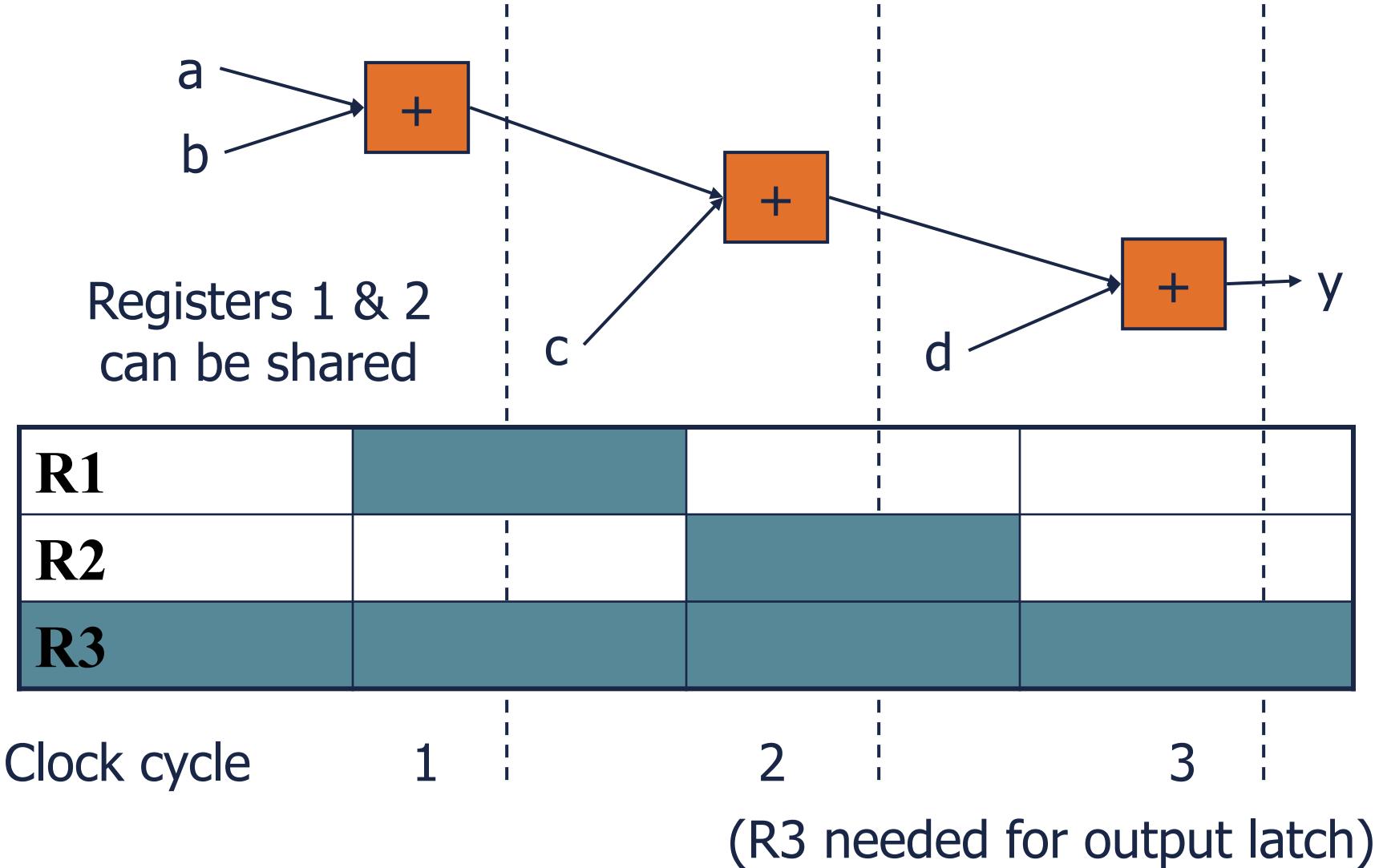
High-Level Synthesis Steps



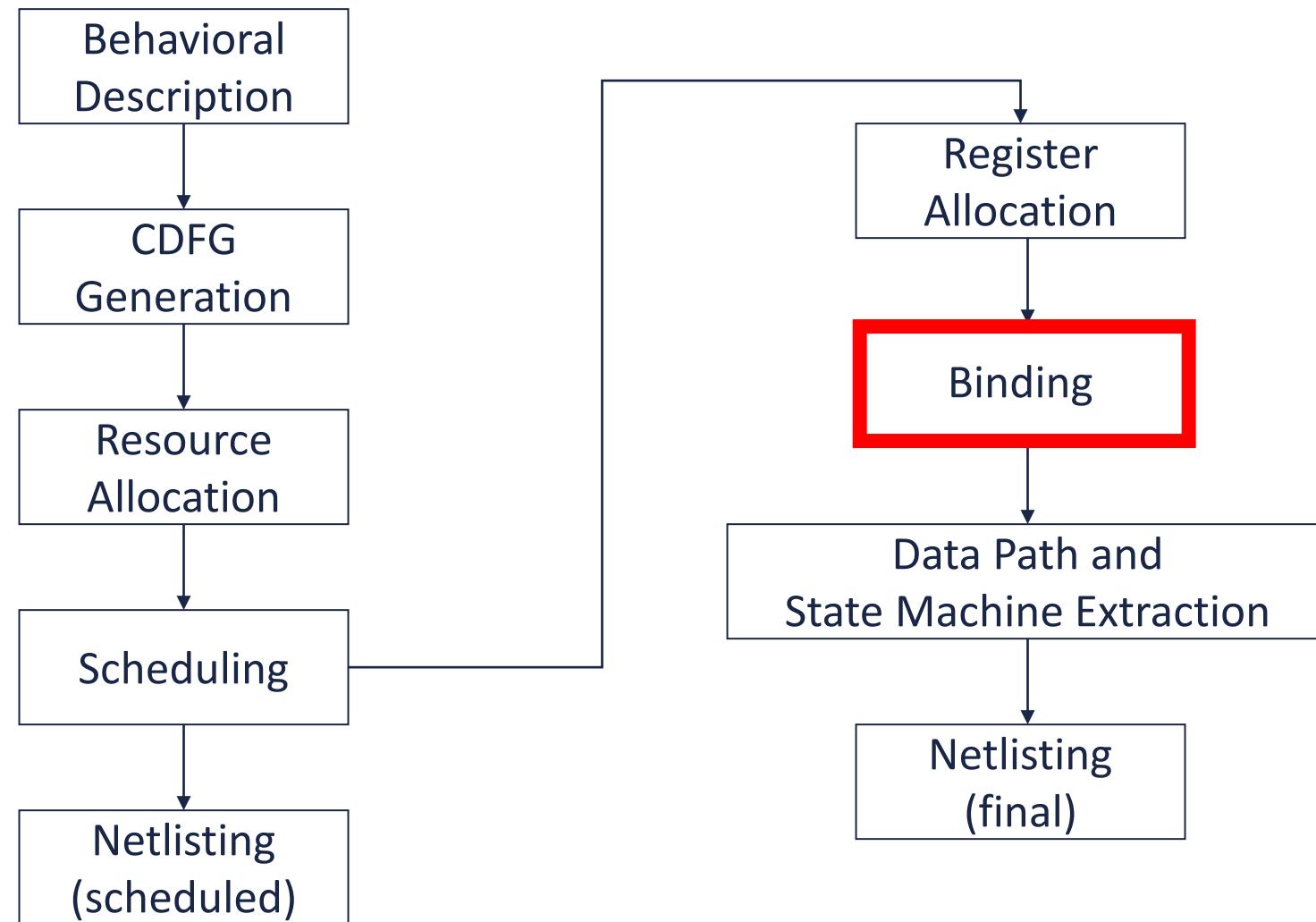
Register Allocation



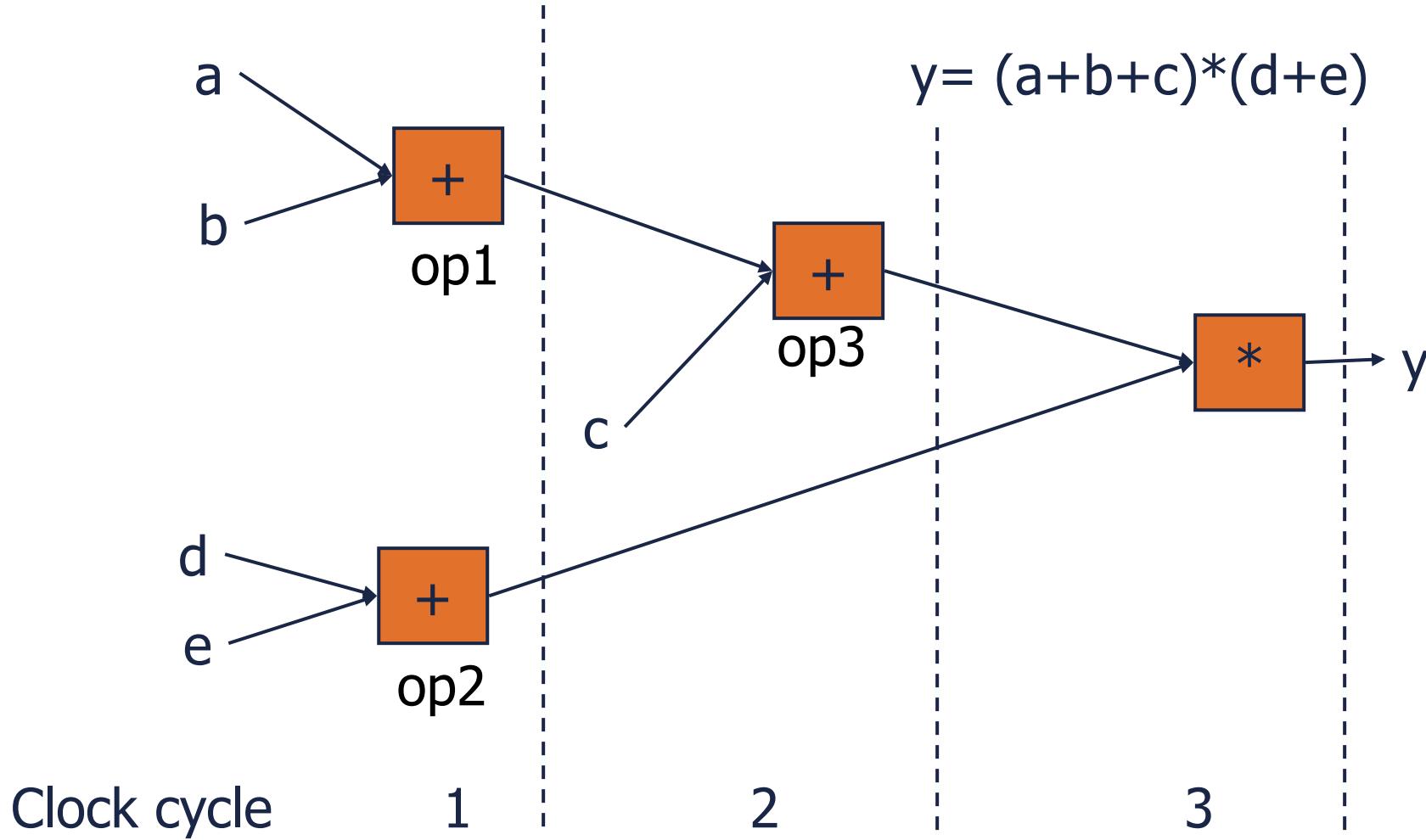
Lifetime Analysis



High-Level Synthesis Steps



Binding



Binding Choices

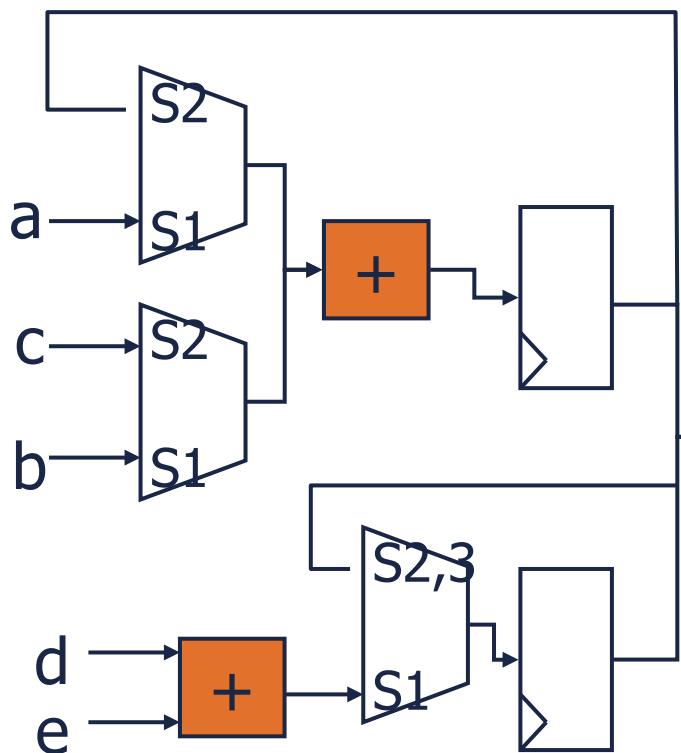
Binding 1

Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add1

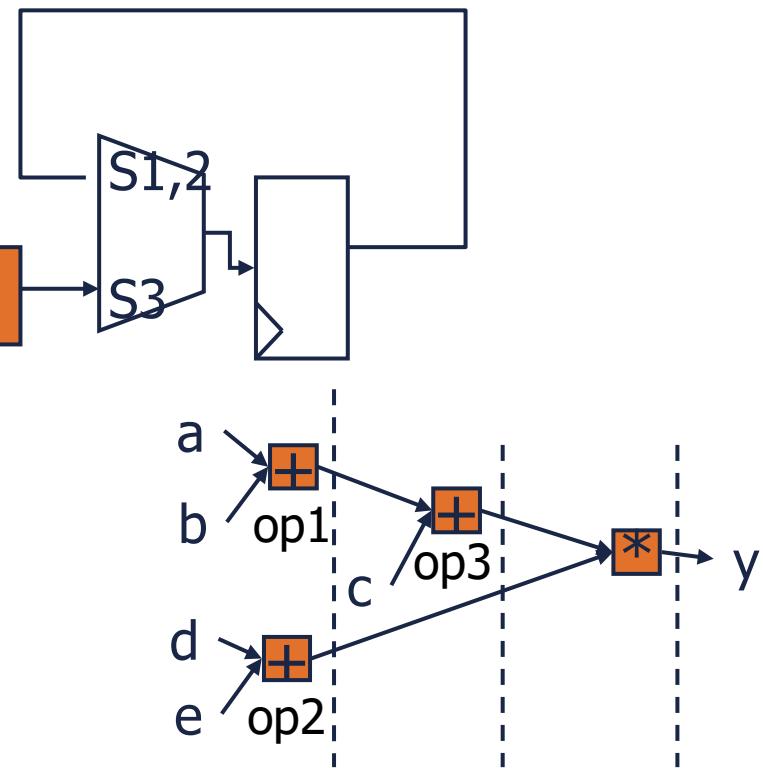
Binding 2

Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add2

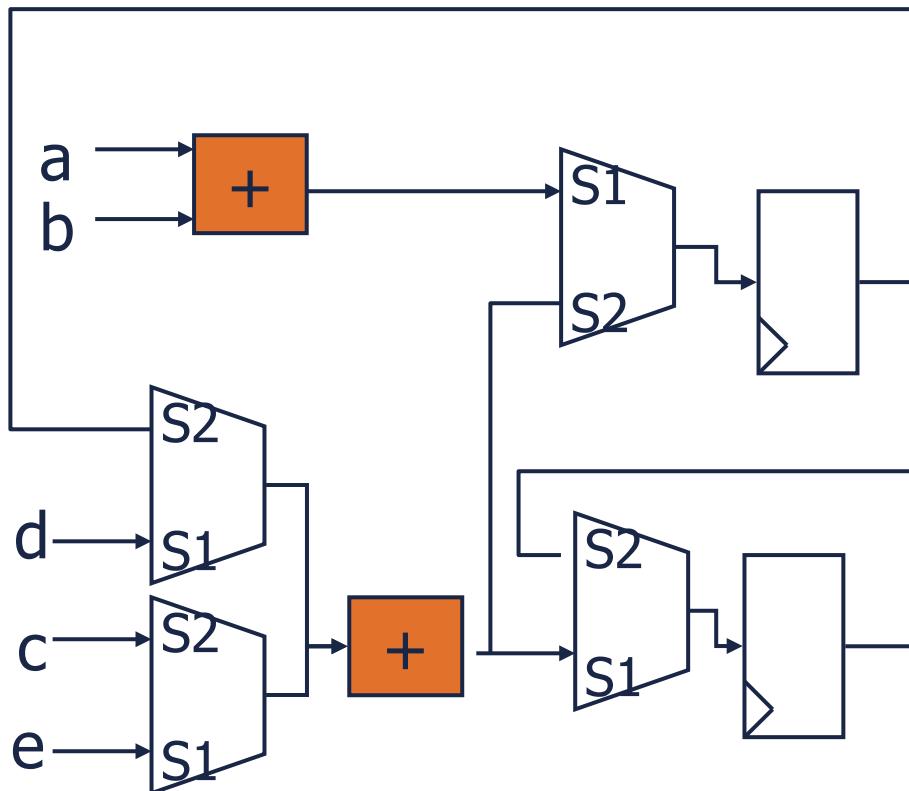
Binding 1 Results



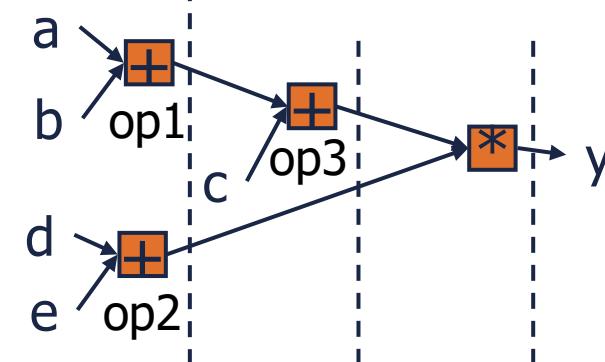
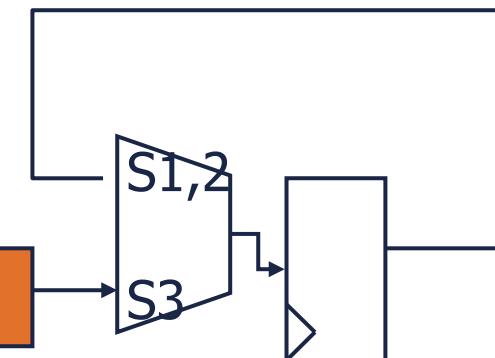
Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add1



Binding 2 Results



Operation	Binding
Op1	Add1
Op2	Add2
Op3	Add2



Linear Programming

- Objective function: a linear function to be maximized or minimized, e.g.,
 - Maximize $c_1x_1 + c_2x_2$
- Constraints: e.g.,
 - $a_{11}x_1 + a_{12}x_2 \leq b_1$
 - $a_{21}x_1 + a_{22}x_2 \leq b_2$
 - $a_{31}x_1 + a_{32}x_2 \leq b_3$
- Non-negative variables, e.g.,
 - $x_1 \geq 0$
 - $x_2 \geq 0$
- The problem is usually expressed in *matrix form*, and then becomes:
 - Maximize $\mathbf{c}^T \mathbf{x}$
 - Subject to $\mathbf{A}\mathbf{x} \leq \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$

Integer Linear Programming (ILP)

- If the variables are all required to be integers, then the problem is called an integer programming (IP) or integer linear programming (ILP) problem.
- In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in the worst case NP-hard.
- 0-1 integer programming is the special case of integer programming where variables are required to be 0 or 1 (rather than arbitrary integers). This method is also classified as NP-hard, and in fact the decision version was one of Karp's 21 NP-complete problems.

ILP-Based Scheduling

- Use binary decision variables
 - $X = \{x_{il}; i = 1, \dots, n; l = 1, \dots, L\}$
 - $x_{il} \in \{0, 1\}$
- First the start time of each operation is unique

$$\sum_l x_{il} = 1, i = 1, 2, \dots, n$$

The start time of any operation v_i :

$$t_i = \sum_l l \cdot x_{il}$$

ILP-Based Scheduling: Constraints

- Given the CDFG represented by $G(V, E)$ d_j : latency of operation v_j
 $t_i \geq t_j + d_j \quad \forall i, j : (v_j, v_i) \in E$ implies

$$\sum_l l \cdot x_{il} \geq \sum_l l \cdot x_{jl} + d_j \quad i, j = 1, 2, \dots, n : (v_j, v_i) \in E$$

- Resource bounds must be met at every schedule time step.

$$\sum_{i: T(v_i)=k} \sum_{m=l-d_i+1}^l x_{im} \leq a_k, \quad k = 1, 2, \dots, n_{res}, \quad l = 1, 2, \dots, L$$

ILP-Based Scheduling: Formulation

- Denote t the vector whose entries are the start times
 - Minimize $C^T t$ such that

$$\sum_l x_{il} = 1, i = 1, 2, \dots, n$$

$$\sum_l l \cdot x_{il} - \sum_l l \cdot x_{jl} - d_j \geq 0, i, j = 1, 2, \dots, n : (v_j, v_i) \in E$$

$$\sum_{\substack{i: T(v_i)=k}} \sum_{m=l-d_i+1}^l x_{im} \leq a_k, k = 1, 2, \dots, n_{res}, l = 1, 2, \dots, L$$

$$x_{il} \in \{0, 1\}, i = 1, 2, \dots, n, l = 1, 2, \dots, L$$

ILP-Based Scheduling: Vector c

- Minimize $c^T t$

The start time of any operation v_i :

$$t_i = \sum_l l \cdot x_{il}$$

- $c = [0, \dots, 0, 1]^T$
 - Minimize the latency of the schedule
- $c = [1, \dots, 1, 1]^T$
 - Finding the earliest start times of all operations

What designs fit HLS well?

- Data path, uni-directional data streaming pipeline
- Flexible in latency requirement, loose timing relationship between interfaces
- Computational kernels
 - E.g., image/video processing/arithmetic computation
 - Dataflow design is better than control flow design
 - HLS tool may provide IPs for arithmetic and other typical computational operations
- No inter-iteration dependencies
 - Preferably no dependencies between two loop iterations
 - Advanced HLS tool can handle dependencies to certain extent
- Independent kernels and module level designs
 - Not top level/system level designs entirely for HLS

Challenges for HLS

- May not be a good fit for very high speed design
 - Designers can push until the last FF and create deep pipelines with RTL
- Cycle accurate design, strong timing relationship on interfaces
- Design with feedback loops and strong timing requirement
 - E.g., accumulators whose current result affects next cycle accumulation immediately
- Control-intensive logic -- but is getting better
- HLS may need designer's manual intervention in order to generate hierarchical designs with high quality
- Coarse-grain pipelining
 - Handling multiple kernels at the top level can be a challenge
 - Advanced HLS tools can work adequately in this area (an example to follow)
- Complicated protocol handling
- Analog/mixed signal portions of the design

Where are these challenges from?

- HLS works with a design entry at a higher abstraction level
 - C/C++ has no idea of “time”
 - It might be do able to capture some timing concept, but the efficiency drops greatly, and we lose the benefit not worth it.
- RTL, as well as SystemC, has “time” built in
 - Fits design with accurate timing requirement best

Coding with HLS

- Write code with sub functions corresponding to modules
 - With hardware implementation concepts in mind
 - Think about the inter-function communication scheme
- Generate hardware for each function using HLS, leave the arguments as interface/ports
- Generate the glue logic/communication and top-level design
 - Integrate the hardware block generated from HLS into a system (e.g., a virtual platform)
 - If the hardware is hierarchical, to achieve better QoR, manual work may be required

Software vs. Hardware Compilation

C Construct	Example	Software Compilation	High-level Synthesis
Function	<code>void go() { ... }</code>	groups of instructions	hardware module
Function args and return	<code>int det(int a[16][16]);</code>	pushed onto call stack	input and output ports
Function call	<code>go();</code>	call instructions	submodule
Operations	<code>prod = a.x * b.x + a.y * b.y;</code>	computation instructions	functional units
Local variables	<code>unsigned index; int sum;</code>	architectural registers	physical registers
Arrays	<code>int A[16][16];</code>	stack, heap, static memory	memory blocks
Control flow	<code>for (i = 0; i < 16; i++) { ... }</code>	branch instructions	state machines
source code	all of the above	machine instructions	hardware modules

C/C++ Code Synthesizability for HLS

- In HLS, your code is the hardware specification

Unsynthesizable C Construct	Example	High-level Synthesis?	Synthesizable alternative
Variable memory allocation	<code>A = malloc(rows * cols * sizeof(A[0][0]));</code>	resizable RAM?	<code>int A[max_rows][max_cols];</code>
Recursion	<code>int bin_search(Array array) { ... bin_search(array); ... }</code>	infinitely nested modules?	use iterative form of binary search
Indirect calls /function pointers	<code>typedef void Callback(); void go(Callback callback) { ... callback(); ... }</code>	dynamically defined submodule?	inline or create “go” function for each callback
Operating system calls	<code>file = fopen(filename, "r");</code>	I/O? Interrupt?	use input ports for streaming in data

C/C++ Code Synthesizability for HLS (cont'd)

Construct	Example	Synthesizable?
Debug output	<code>printf("sum: %x\n", sum);</code>	Can be translated to unsynthesizable RTL debug functions.
Time functions	<code>clock_gettime(CLOCK_REALTIME, &cur_time);</code>	Not synthesizable. HLS tools often have profiling support built-in.
C++ memory allocation	<code>Node * node = new Node();</code>	Equivalent to malloc; not synthesizable.
C++ template library	<code>std::vector<int> a; ... std::sort(a.begin(), a.end());</code>	Uses dynamic memory allocation and recursion; not synthesizable.
Floating point operations	<code>float a, b, c; ... float d = a * b + c;</code>	Supported by some HLS engines. Can be expensive, especially on FPGAs.

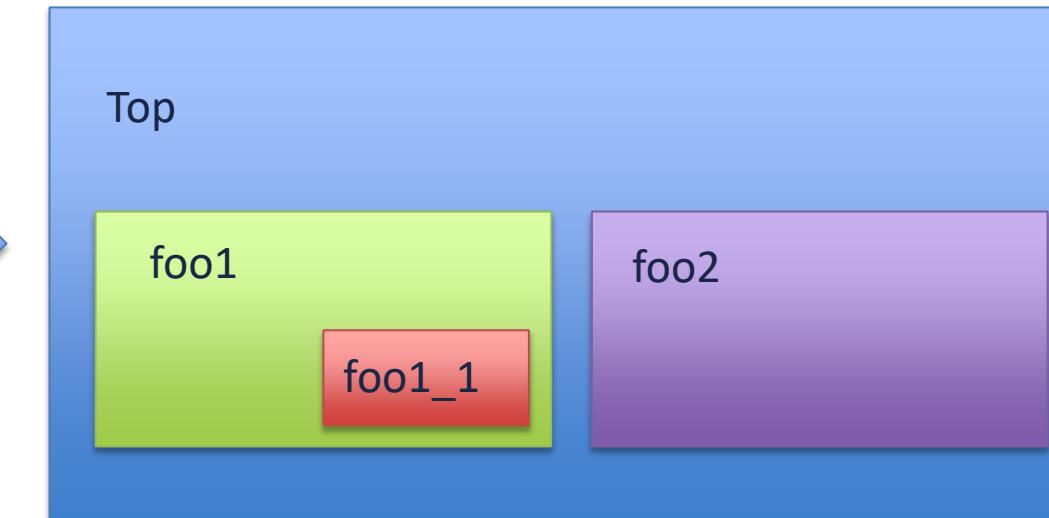
Variables

- Customize bit-widths (arbitrary bit-widths) instead of standard C data types
 - E.g., use 24-bit width instead of 32 bits if 24 bits precision is enough
 - Most HLS tools support arbitrary bit-width data types
 - E.g., ap_int, ap_fixed data types in Vivado HLS
 - Improves efficiency, performance, and saves area

Functions

- C functions are synthesized to RTL modules
 - By default, sub functions become sub modules
 - Usually, every function call is an instance of the RTL module
 - Inline function is an exception – it flattens the design

```
void top(int a, int b) {  
    foo1(a);  
    foo2(b);  
}  
void foo1(int a) {  
    foo1_1(a);  
}
```



Functions: Splitting

- Use subfunctions to improve parallelism
 - Use function to maintain hierarchy
 - Most tools support dataflow/block-level pipelining and parallelism
 - May not be feasible due to dependencies

```
void top (int a[100], int b[100]) {  
    for(i=0; i<100; i++) {  
        a[i] = i;  
    }  
    for(j = 0; j < 100; j++) {  
        z += b[j];  
    }  
}
```

phase 1

phase 2

Original

loop0

loop1

Splitting

foo1
foo2

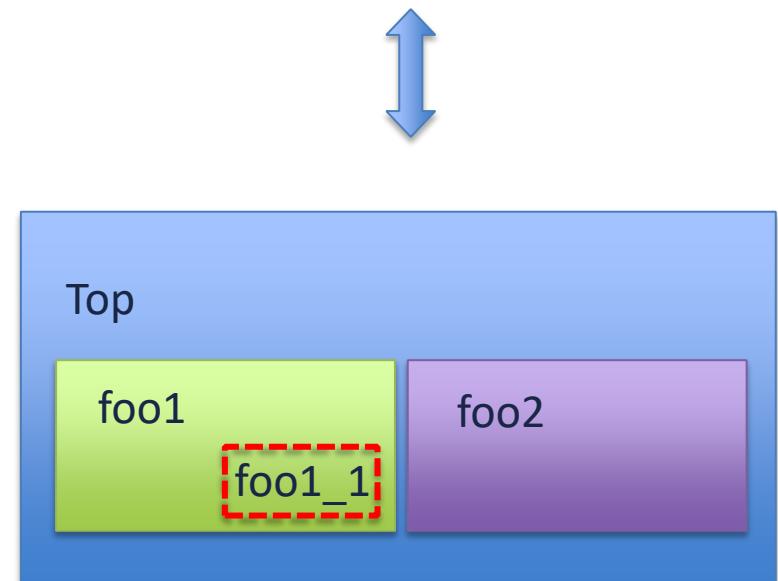


```
void foo1 (int a[100]) {  
    for(i=0; i<100; i++) {  
        a[i] = i;  
    }  
}  
void foo2 (int b[100]) {  
    for(j = 0; j < 100; j++) {  
        z += b[j];  
    }  
}  
void top (int a[100], int b[100]) {  
    foo1(a);  
    foo2(b);  
}
```

Functions: Inlining

- Inline functions to flatten the hierarchy
- Pros:
 - Enable more optimization opportunities
 - Creates more objects to schedule and optimize
 - Enable resource sharing
 - Eliminates function call overhead
- Cons:
 - Module instance is effectively copied for each inlined call
 - Inlining of multiple function calls increases area cost

```
void top(){int a, int b){  
    foo1(a);  
    foo2(b);  
}  
void foo1(int a){  
    foo1_1(a);  
}  
inline void foo1_1(int a){}
```



Inlining: Example

- Merge foo1 and foo2, and apply loop pipelining

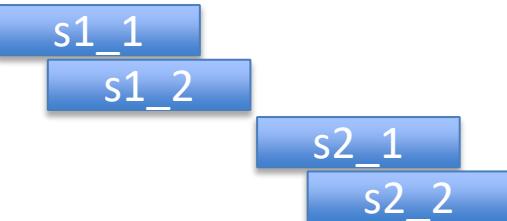
```
int foo1(int a, int b, int c) {  
    z0 = a * b;          //s1_1  
    z1 = c * b;          //s1_2  
    return (z0+z1);  
}  
int foo2(int c, int d, int a){  
    z2 = c * d;          //s2_1  
    z3 = d * a;          //s2_2  
    return (z2+z3);  
}  
int top(int a, int b, int c, int d){  
    y1 = foo1(a, b, c);  
    y3 = foo2(c, d, a);  
    return (y3+y1);  
}
```



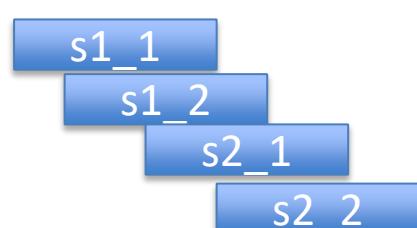
```
int top(int a, int b, int c, int d){  
    z0 = a * b;          //s1_1  
    z1 = c * b;          //s1_2  
    z2 = c * d ;         //s2_1  
    z3 = d * a;          //s2_2  
    return (z0+z1+z2+z3);  
}
```

Assume each multiplication takes multiple cycles, and it can be pipelined

Original



Inlining



Also, enable resource sharing, e.g. for the multiplier

Loops: Bounds

- Not recommend using variable loop bounds, because:
 - HLS tools may not be able to estimate the performance
 - Limits the optimization opportunities
 - E.g., cannot deliver complete loop unrolling, thus prevent the pipelining
 - Solutions: Set the bounds to the maximum value, and conditionally execute the loop body

```
for (i=0; i < len; i++) {  
    A[i] += b;  
}
```

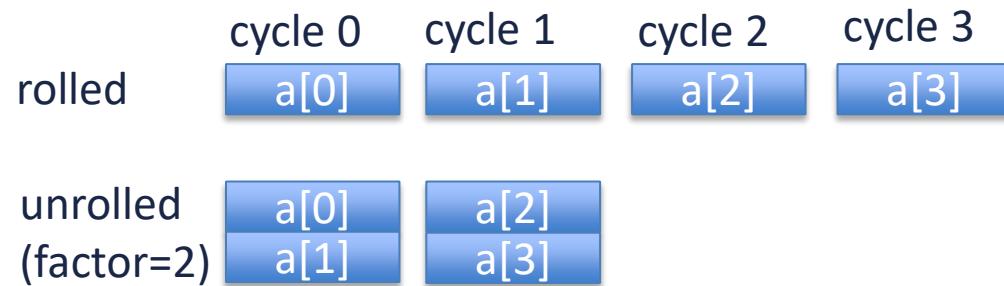


```
for (i=0; i < N-1; i++) {  
    if(i < len)  
        A[i] += b;  
}
```

Loop Unrolling

- Most HLS tools support automatic loop unrolling (by using HLS unroll pragmas)
 - But have strict limits, e.g., constant loop bounds
- May need manual loop unrolling to
 - Improve pipeline opportunity
 - Improve parallelism
- Things to notice
 - Need to consider memory port limit when accessing array
 - Expensive hardware replication
 - Need to balance the unroll level

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



Loop Fusion/Merging

- Merge several loops into one loop
- Some tools support loop fusion
 - Strict bound check of loops, e.g., identical loop bounds
- Manually loop fusion can be used to
 - Improve data locality
 - Shorten latency
 - Share resources

```
void top(){  
    for(i=0; i<N; i++){  
        a[i] = i;  
    }  
    for(i=1; i<N; i++){  
        x+=a[i-1];  
    }  
}
```



```
void top(){  
    for(i=0; i<N; i++){  
        a[i] = i;  
        if(i>=1)  
            x+=a[i-1];  
    }  
}
```

After fusion, loop bodies are optimized together

Loop Perfectization

- Convert loops to perfect loop
 - All statements are in the innermost loop
 - An example: Loop Sinking

```
for (i=0, i<20; i++){  
    a = 0;  
    for(j=0; j<20; j++){  
        a += A[j] + j;  
    }  
    B[i] = a * 20;  
}
```

```
for (i=0, i<20; i++){  
    for(j=0; j<20; j++){  
        if(j==0) a =0;  
        a += A[j] + j;  
        if(j==19) B[i] = a*20; }  
}
```



```
Loop_I_Loop_J:  
    *Trip count: 400  
    *Latency: 401  
    *Pipeline II: 1  
    *Pipeline depth: 2
```



HLS result (Use Vivado_HLS):

```
Loop_I:  
    *Trip count: 20  
    *Latency: 480  
Loop_J:  
    *Trip count: 20  
    *Latency: 21  
    *Pipeline II: 1  
    *Pipeline depth: 2
```

The original code can only pipeline Loop_J. After perfection, two loops can be flattened and pipelined.

Why Loop Perfectization?

- Form very regular loop structure
 - Some tools can only flatten the perfect loops
 - Easy for HLS scheduling and discovering optimization opportunities
 - Improves performance by eliminating the loop entering/exiting/cost
- Different ways of loop perfection
 - Loop sinking
 - Loop unrolling
 - Loop splitting

Loop Perfectization: Code Sinking

```
for (i=0, i<20; i++){  
    a = 0;  
    for(j=0; j<100; j++){  
        a += A[j] + j;  
    }  
    B[i] = a * 20;  
}
```



```
for (i=0, i<20; i++){  
    for(j=0; j<20; j++){  
        if(j==0) a = 0;  
        a += A[j] + j;  
        if(j==19) B[i] = a*20;  
    }  
}
```

- Pros:
 - Easy to implement
 - No performance penalty for branch operations in pipeline
- Cons:
 - Expensive for hardware
 - sometimes the branches can be messy when code is complicated
 - Could sequentialize loop nest and affect performance
 - Change the data dependency

Loop Perfectization: Loop Unrolling

```
for (i=0; i<100; i++) {  
    z = a*b[i];  
    z+= c*d[i];  
    for(j=0; j<4; j++) {  
        z+=x[j] + j;  
    }  
}
```



```
for (i=0, i<100; i++) {  
    z = a*b[i];  
    z += c*d[i];  
    z += x[0] + 0;  
    z += x[1] + 1;  
    z += x[2] + 2;  
    z += x[3] + 3;  
}
```

- Pros:
 - Enable pipelining at outer loop
 - Enable resource sharing
 - More statements to schedule in the same loop level
 - Higher performance, good for expensive operations
- Cons:
 - Not always feasible
 - Expensive hardware, best if the innermost loop is small

Loop Perfectization: Loop Splitting

```
for (i=0; i<100; i++) {  
    z = a*b[i];  
    z += c*d[i];  
    for(j=0; j<4; j++){  
        z+=x[i] + j;  
    }  
}
```



```
for(i=0; i<100; i++) {  
    z = a*b[i];  
    z += c*d[i];  
}  
for(i=0, i<100; i++) {  
    for(j=0; j<4; j++) {  
        z += x[0] + 0;  
        z+= x[1] + 1;  
        z+= x[2] + 2;  
        z+= x[3] + 3;  
    }  
}
```

- Pros:
 - Enable optimization per loop
 - Opportunity for dataflow streaming between two loops
- Cons:
 - Not always possible because of data dependency

Arrays

- By default, it is implemented as RAM/ROM
- Expensive and have limited ports
- Optimizations:
 - Array streaming
 - Replace the RAM with FIFO, if possible
 - Requires array access to be sequential
 - Requires the array access order to be the same
 - Transform the code to make array access sequential
 - Array partitioning
 - Eliminate the ports limit

Array Streaming: Example

```
for (i=0; i<100; i++)  
    for(j=0; j<100; j++){  
        A[i][j] = a * b; //row  
        ...  
    }  
}  
  
for (i=0; i<100; i++)  
    for(j=0; j<100; j++) {  
        C[i][j] = A[j][i]; //col  
    }  
}
```

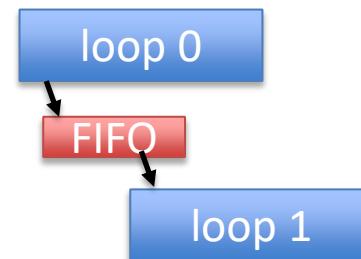


```
hls::stream<int> A;  
for (i=0, i<100; i++){  
    for(j = 0; j<100; j++)  
        A.write(a*b);  
}  
for(i=0, i<100; i++){  
    for(j = 0; j<100; j++){  
        A.read(C[j][i]);  
    }  
}
```

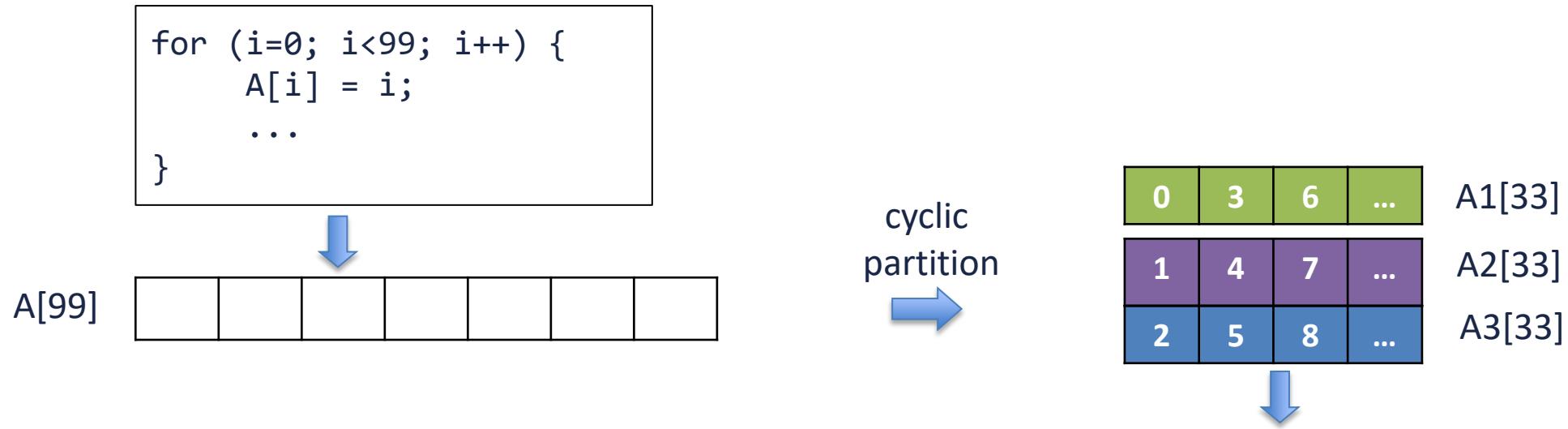
- Original array A access order:
 - Loop1: row; Loop2: column
 - No dataflow pipeline
 - No streaming



- New array A access order:
 - Loop1: row; Loop2: row
 - Dataflow pipeline
 - Streaming



Array Partitioning



- To mitigate the # of ports limitation
- To improve parallelism
- Be careful:
 - sometimes array partition can be subtle
 - can easily introduce bugs

```
for (i=0; i<99; i+=3) {  
    A1[i] = i;  
    A2[i] = i+1;  
    A3[i] = i+2;  
    ...  
}
```

Some Tips for C-based HLS Design

- Understand the HLS tool behavior and the optimization goal
 - No unique coding style is “useful” or “good” for all optimizations
- Reuse data and minimize array access
 - Once the data has been read into a block or buffer and reused, it can improve the parallelism.
 - Array ports are limited. Array partitioning can be an alternative, but it’s not free.
- Loop transformation to enable parallelism
 - Pipelining/Unrolling/Dataflow
- Bit-width selection for efficient hardware
- The optimization goals for CPU and Hardware are different
 - E.g., branch mis-prediction
 - But many software code optimization techniques are useful for hardware design too

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 6:

Compilers for Accelerators

Sitao Huang

sitaoh@uci.edu

February 15, 2022



Logistics

- Please submit your project proposal ASAP
 - Options: (a) literature review paper or (b) compiler + accelerator project
- Project option (a) or option (b)
 - Reasonable workload for 3 weeks
 - Workload will be considered in evaluation

Tentative Schedule

- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7 (2/15, 2/17): Compilers for Accelerators**
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow



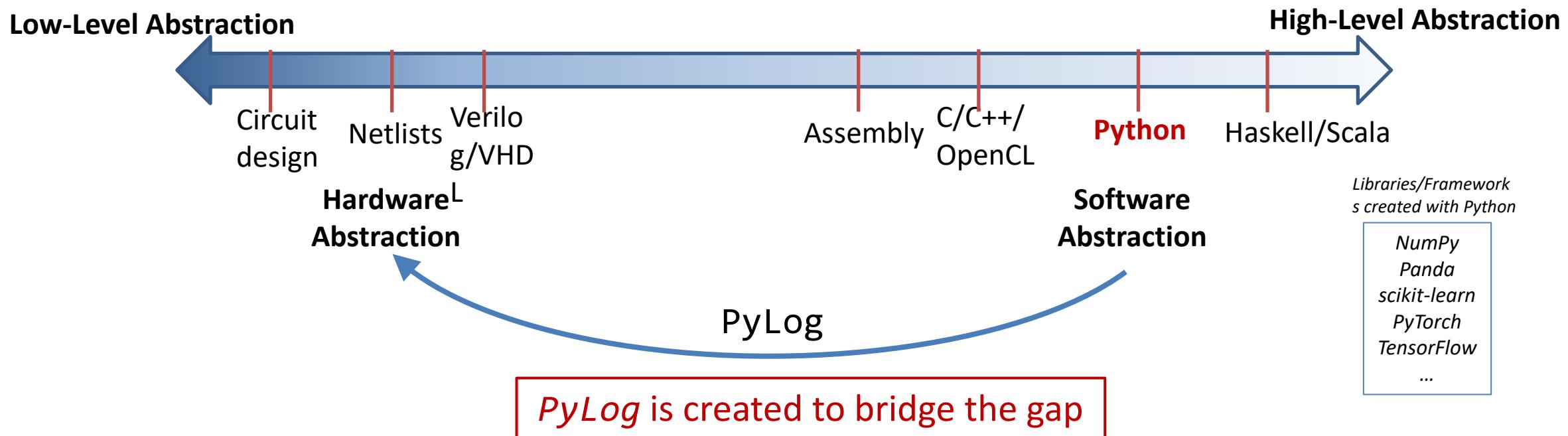
<https://github.com/hst10/pylog>



<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9591456>

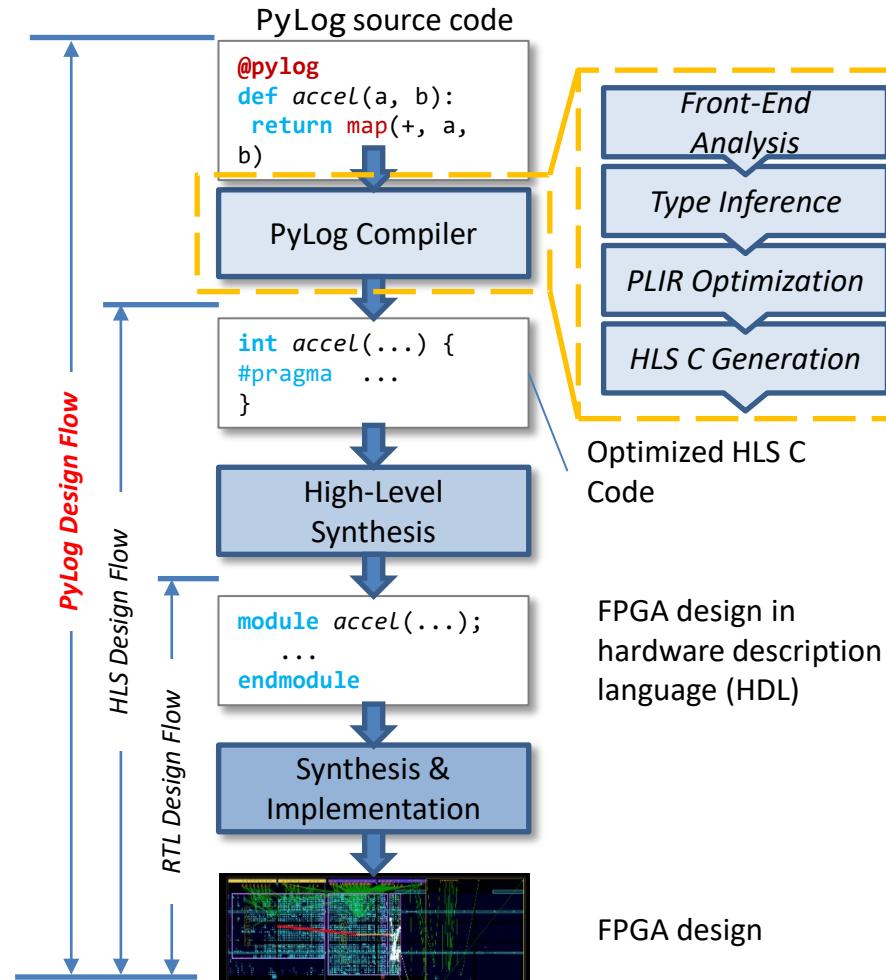
Gap between Hardware and Software Abstraction Levels

- Huge *gap* between *hardware* abstraction levels and *software* abstraction levels
- *Large* number of applications/frameworks are developed with *high-level* languages
- Many applications in high-level abstraction need *hardware acceleration*
- *Very challenging* to accelerate high-level applications due to abstraction gap



PyLog: High-Level Programming and Synthesis Flow for FPGAs

FPGA Programming is moving towards higher abstractions



High-level synthesis (HLS) flow greatly improves design productivity:

- Fewer lines of code (LOC)
- Easier system integration (e.g. cloud/edge FPGAs)
- Easier design space search, explore more alternative designs, achieve better QoR
- Easier design migration
- ...

Some drawbacks:

- Limitations in expressing concurrency with C semantics
- Conservative compiler optimizations due to the lack of high-level computation pattern information
- Highly performant code requires careful tuning and LOC increases quickly (e.g. 8,000 lines of HLS C code for optimized convolution kernel)
- ...

Benefits of using high-level abstraction (C language)

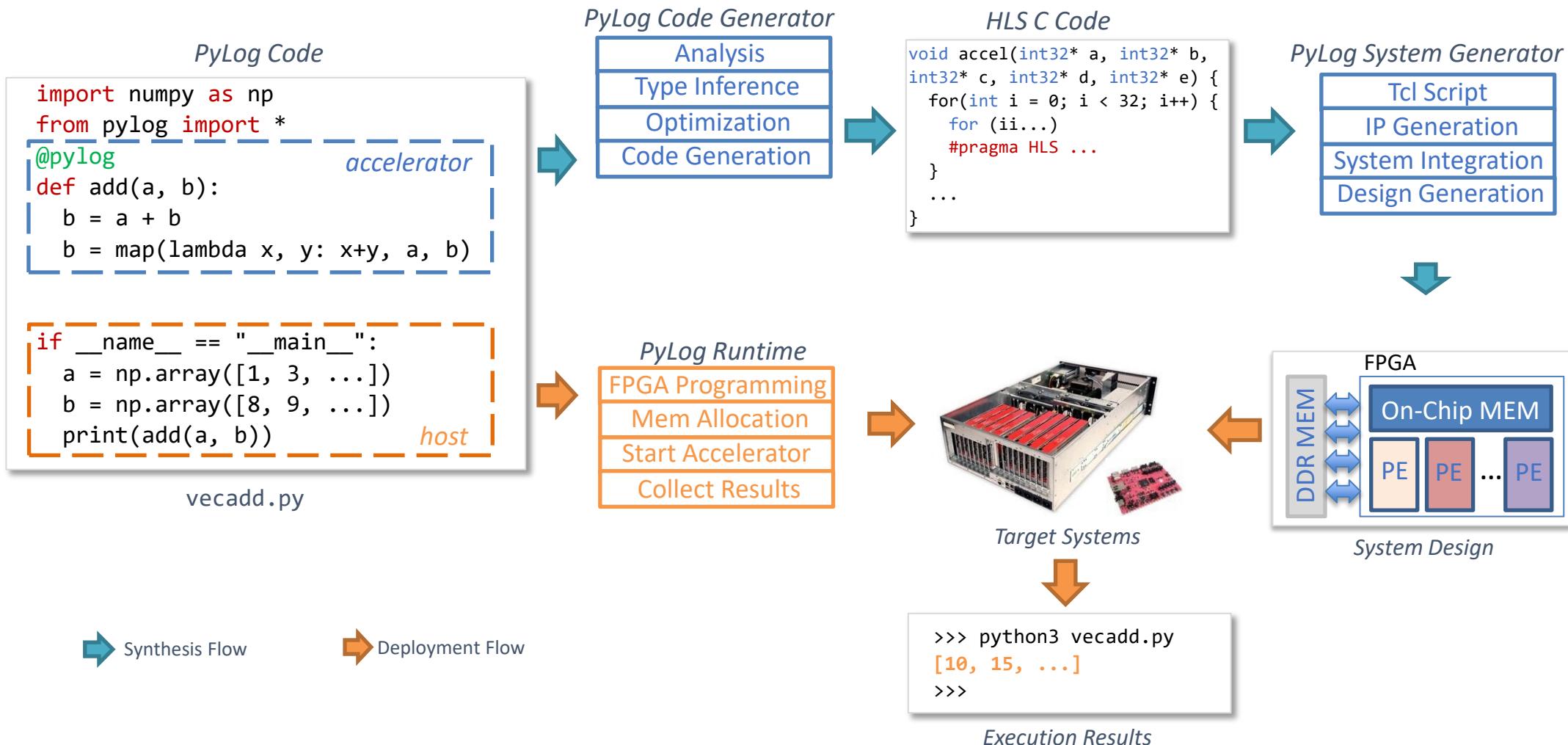
Need high-level programming and synthesis flow!

Need languages with high-level operators (high-level languages)

Need powerful compiler optimizations

Easier to achieve with high-level languages

PyLog Flow Overview



<https://github.com/hst10/pylog>

PyLog Language and Compiler Highlights

High-level operators, design space exploration

```
1D Conv: y = map(Lambda a: a[-1]+a[0]+a[1], x[1:-1])
2D Conv: y = map(Lambda a: dot(a[-1:2, -1:2], w), img[1:-1, 1:-1])
```

Multiple possible C code versions from high-level code

```
for (...) {  
#pragma HLS  
unroll  
}
```

```
for (...) {  
#pragma HLS unroll factor=4  
}
```

```
for (...) {  
#pragma HLS  
pipeline  
}
```

PyLog chooses the optimal implementation given the HW resource of the target platform

Python statements, compute customization

```
@pylog
def pl_matmul(a, b, c, d):

    buf = np.empty([16, 16], pl_fixed(8, 3))
    pragma("HLS array_partition variable=buf")

    def matmul(a, b, c):
        for i in range(32):
            for j in range(32).unroll(4):
                tmp = 0.0
                for k in range(32).pipeline():
                    tmp += a[i][k] * b[k][j]
                c[i][j] = tmp
```

Unified FPGA and host programming

```
import numpy as np
from pylog import *
```

```
@pylog
def add(a, b):
    return a + b
```

```
if __name__ == "__main__":
    a = np.asarray([1, 3, 6, 7])
    b = np.asarray([8, 9, 10, 5])
```

```
print(add(a, b))
```

accelerator

host

Type inference and type checking

- No explicit type annotation required
- Top function has NumPy arrays as arguments, which carries input type and shape information
- Type engine infers type and shape of each object in the code

pl_type: PLType(float, 0) pl_type: PLType(float, 2)
pl_shape: (0,) pl_shape: (3, 3)

y = map(Lambda a: dot(a[-1:2, -1:2], w), img)

pl_type: PLType(float, 2) pl_type: PLType(float, 0) pl_type: PLType(float, 2) pl_type: PLType(float, 2)
pl_shape: (27, 27) pl_shape: (0,) pl_shape: (3, 3) pl_shape: (27, 27)

(known) (known)

PyLog High-Level Operations

■ map operation: $map(f, x_0, x_1, \dots)$

- Repeatedly apply function f to each element in x_0, x_1, \dots
- Allow access neighbor elements inside function

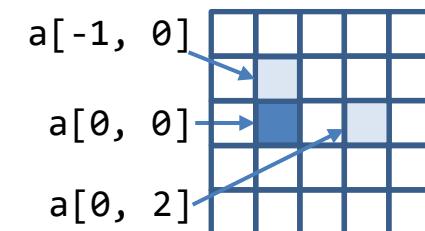
Example: vector addition (a and b have same shape)

```
z = map(Lambda a, b: a + b, x, y)
```

■ Dot operation: dot product of two arrays

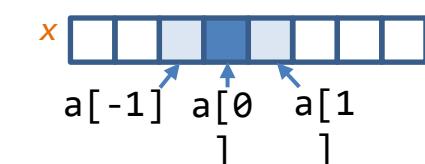
(a) 2D map

```
PyLog   y = map(Lambda a0, a1, ...: op(a0[-1, 0], a1[0, 2], ...), x0, x1, ...)  
HLS C   L1: for(int i1 = 0; i1 < x0.dim(0); i1++) {  
          L2:   for(int i2 = 0; i2 < x0.dim(1); i2++) {  
                  y[i1][i2] = op(x0[i1-1][i2], x1[i1][i2+2], ...); }}
```



(b) 1D conv

```
PyLog   y = map(Lambda a: a[-1]+a[0]+a[1], x[1:-1])  
HLS C   L1: for(int i1 = 1; i1 < x.dim(0)-1; i1++) {  
          y[i1] = x[i1-1] + x[i1] + x[i1+1]; }
```

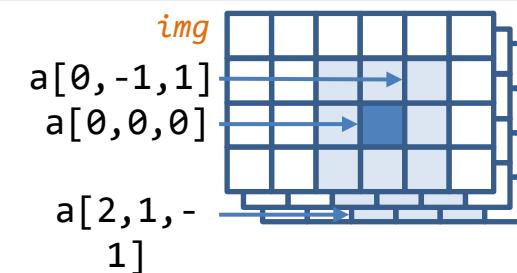


(c) 2D conv (with multiple channels)

```
PyLog   y = map(Lambda a: dot(a[:, -1:2, -1:2], w), img[0, 1:-1, 1:-1])  
(HLS C code omitted for simplicity)
```

3x3 multiplication and reduction across *all channels*

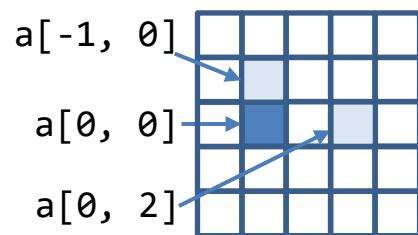
Image without outermost pixels



PyLog High-Level Operations

- map operation: $map(lambda\ a_0, a_1, \dots : \dots, x_0, x_1, \dots)$
 - Repeatedly apply lambda function to each element in x_0, x_1, \dots
 - Allow access neighbor elements inside lambda function
- Dot operation: dot product of two arrays

```
PyLog  y = map(Lambda a0, a1, ...: op(a0[-1, 0], a1[0, 2], ...), x0, x1, ...)  
      L1: for(int i1 = 0; i1 < x0.dim(0); i1++) {  
HLS C  L2:   for(int i2 = 0; i2 < x0.dim(1); i2++) {  
                y[i1][i2] = op(x0[i1-1][i2], x1[i1][i2+2], ...); }}
```



Some other examples:

```
1 # Vector add  
2 out = map(lambda x, y: x + y, vec_a, vec_b)  
3  
4 # 1D convolution  
5 out = map(lambda x:x[-1]+x[0]+x[1], vec[1:-1])  
6  
7 # Inner product  
8 out_vec[i] = dot(matrix[i, :], in_vec)  
9  
10 # Square matrix multiplication  
11 out = map(lambda x,y: dot(x[0,:],y[:,0]), mat_a, mat_b)
```

PyLog Supported Platforms

- Platforms supported by PyLog

FPGA Platform Type	Platforms	Synthesis Flow	Runtime Library
SoCs and MPSoCs	ZedBoard, PYNQ, Ultra96	Vivado HLS + Vivado	PYNQ
PCIe-Based High-End FPGAs	Amazon EC2 F1 instance (XCVU9p), Alveo series (U200, U250, U280)	Vivado HLS + Vitis	PYNQ

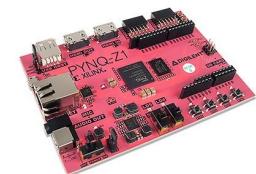
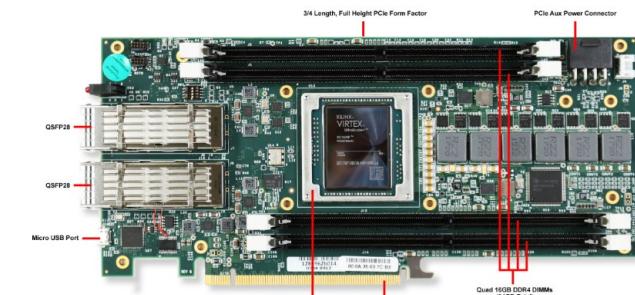
- Design and deploy for different platforms easily
 - Simply change @pylog mode, no extra coding needed

Synthesize for AWS F1: `@pylog(mode='hwgen', board='aws_f1')`

Deploy on AWS F1: `@pylog(mode='deploy', board='aws_f1')`

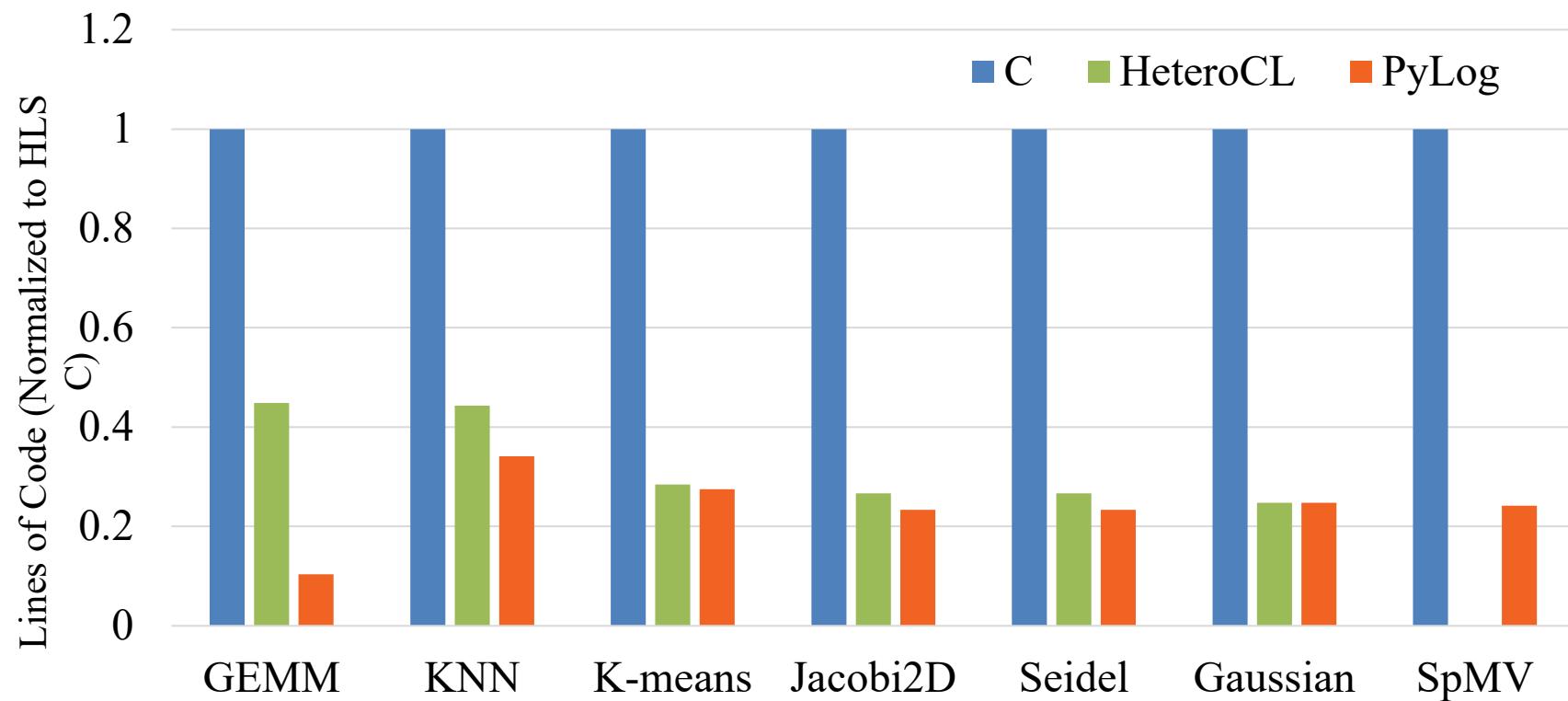
Synthesize for PYNQ: `@pylog(mode='hwgen', board='pynq')`

Deploy on PYNQ: `@pylog(mode='deploy', board='pynq')`



PyLog Evaluation: Expressiveness

- Compare LoC (Lines of Code) of HLS C code vs HeteroCL* code vs PyLog code
 - Normalized to HLS C LOC
 - **70%** reduction compared to HLS C code

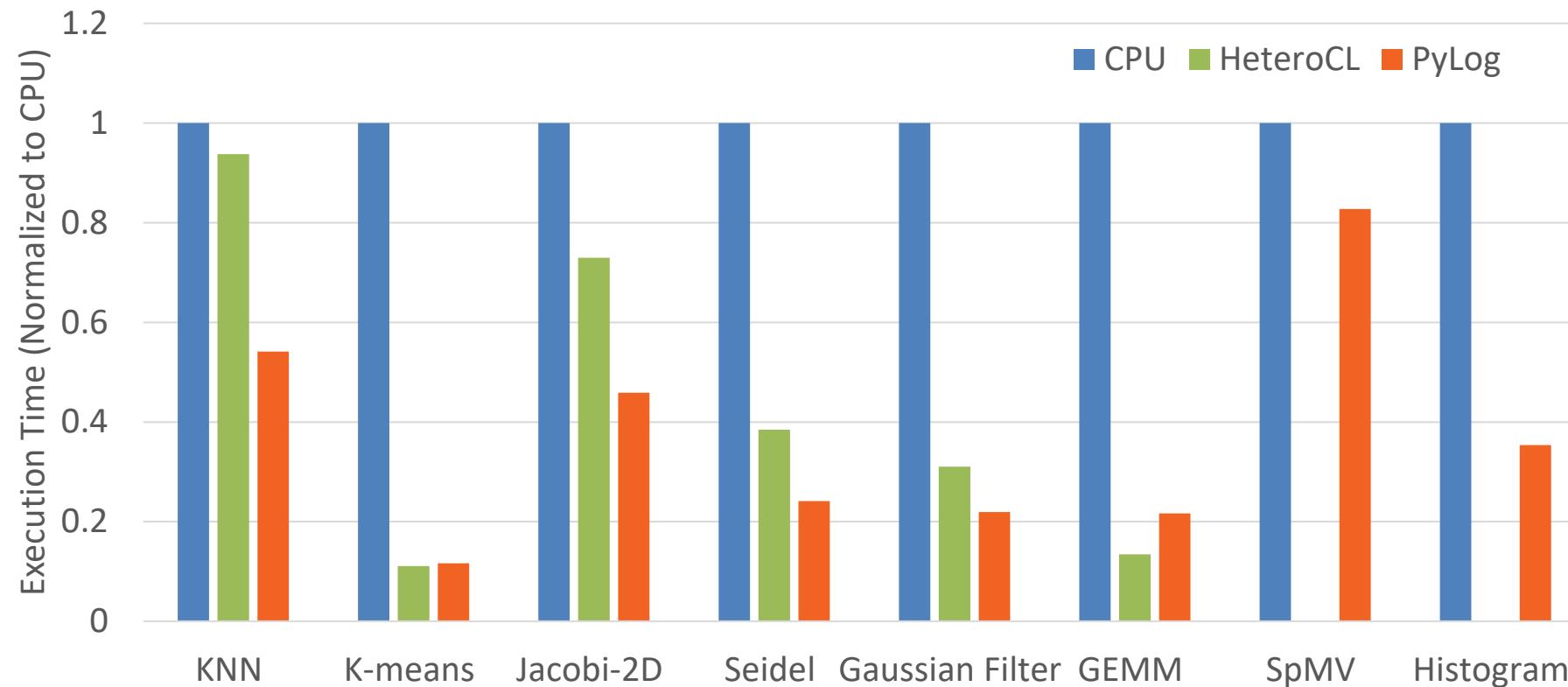


*Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. FPGA 2019.

PyLog Evaluation: Accelerator Performance

Compare accelerator performance vs CPU (C++) performance and HeteroCL accelerator performance

- Platform: Amazon AWS F1 instance (Xilinx Virtex UltraScale+ XCVU9P)
- **3.17x** and **1.24x** faster than optimized CPU and FPGA accelerators

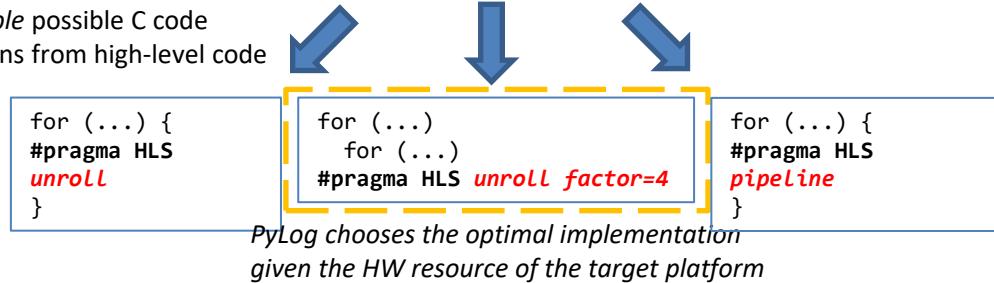


CPU: Optimized CPU code running on single thread on 8-core Intel Xeon E5-2686 v4 CPU

Operator Implementation Selection

```
1D Conv: y = map(Lambda a: a[-1]+a[0]+a[1], x[1:-1])
2D Conv: y = map(Lambda a: dot(a[-1:2, -1:2], w), img[1:-1, 1:-1])
```

Multiple possible C code versions from high-level code



Applied to:

- N -dimensional array operations, e.g. $c = a + b$
- PyLog high-level operations, e.g. map, dot, reduce, etc.

PyLog IR:

- Represent the transformations in IR systematically
- Represent performance and cost of each version

PyLog internal:

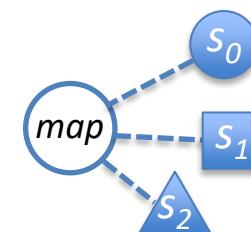
- *PyLog schedule: PLSchedule*
- Sequence of transformations applied to loop nests, subscripts, etc.
- Definition:

```
schedule = [ (action1, arg11, arg12, ...),  
            (action2, arg21, arg22, ...) ]
```
- Example:

```
schedule = [ ('interchange', 1, 3),  
            ('tile', 3, 4) ]
```
- Example (with parameters):

```
schedule = [ ('interchange', 1),  
            ('tile', 3) ]
```
- Extensible: action_object are predefined Python functions to apply action to object, e.g. *interchange_PLSubscript*, *tile_list*, etc.

Each high-level operations will have a list of candidate schedules, `schedules = [s0, s1, s2, ...]`



Operator Implementation Selection

2D Add:

$$c = a + b$$

Plain

```
for1 (...)  
for2 (...)
```

Interchange 1, 2

```
for2 (...)  
for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)  
for11 (...)  
for12 (...)
```

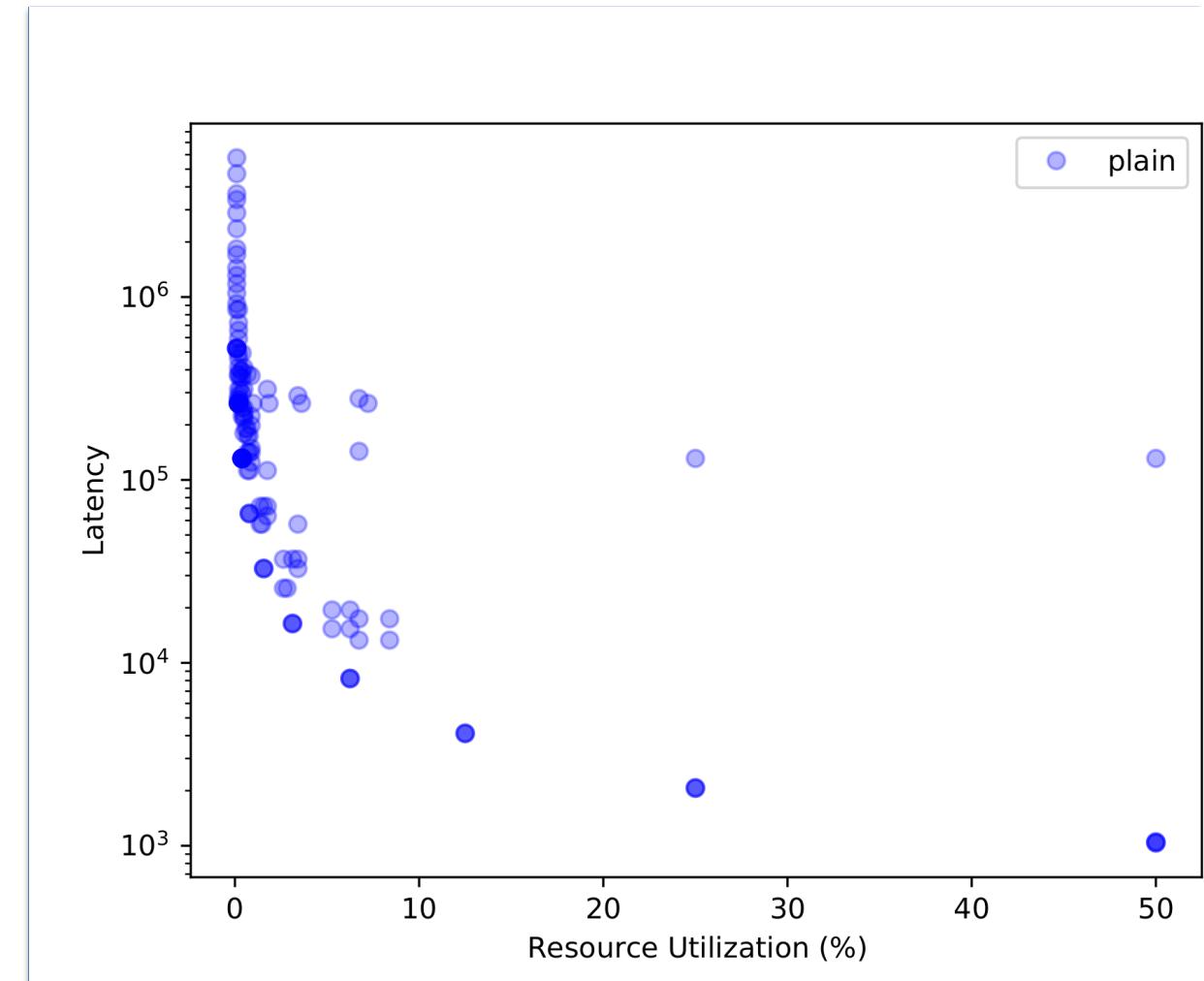
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)  
for22 (...)  
for11 (...)  
for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)  
for11 (...)  
for22 (...)  
for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

$$c = a + b$$

Plain

```
for1 (...)  
for2 (...)
```

Interchange 1, 2

```
for2 (...)  
for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)  
for11 (...)  
for12 (...)
```

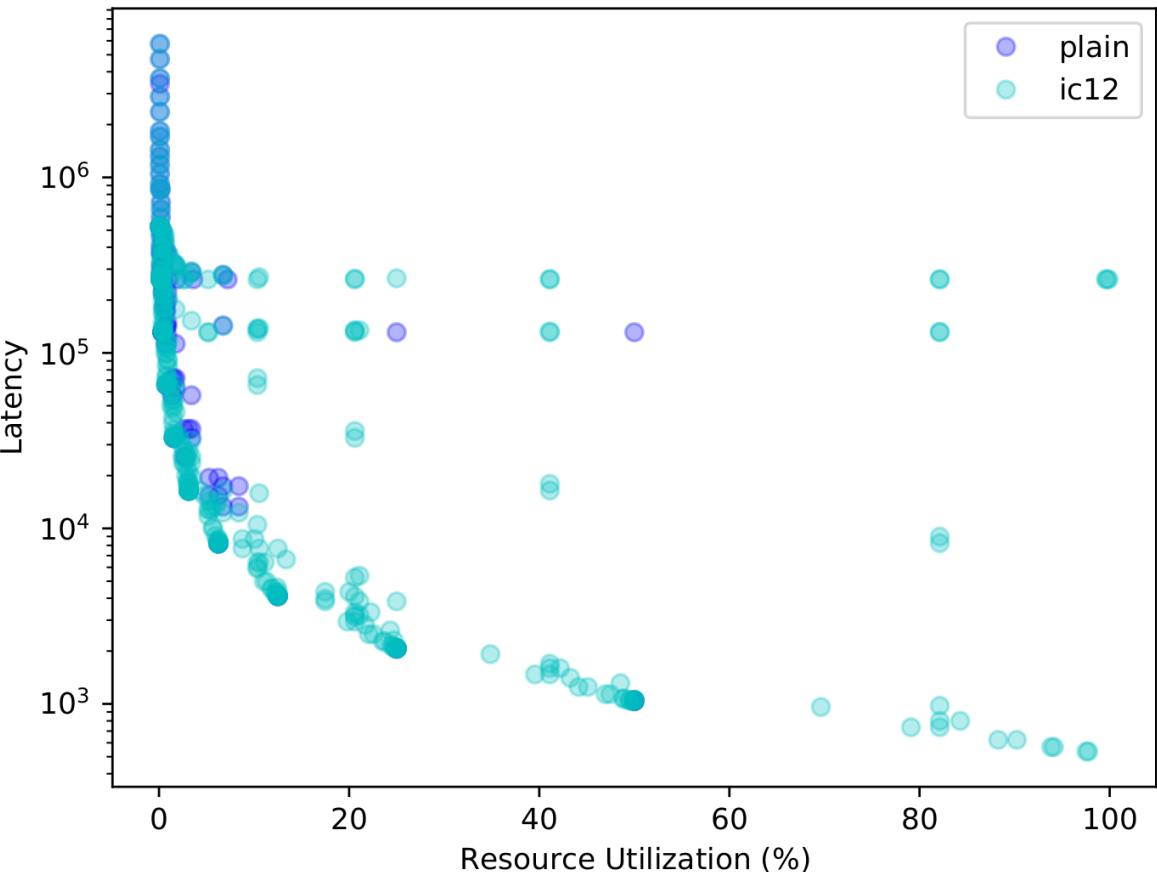
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)  
for22 (...)  
for11 (...)  
for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)  
for11 (...)  
for22 (...)  
for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

$$c = a + b$$

Plain

```
for1 (...)  
for2 (...)
```

Interchange 1, 2

```
for2 (...)  
for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)  
for11 (...)  
for12 (...)
```

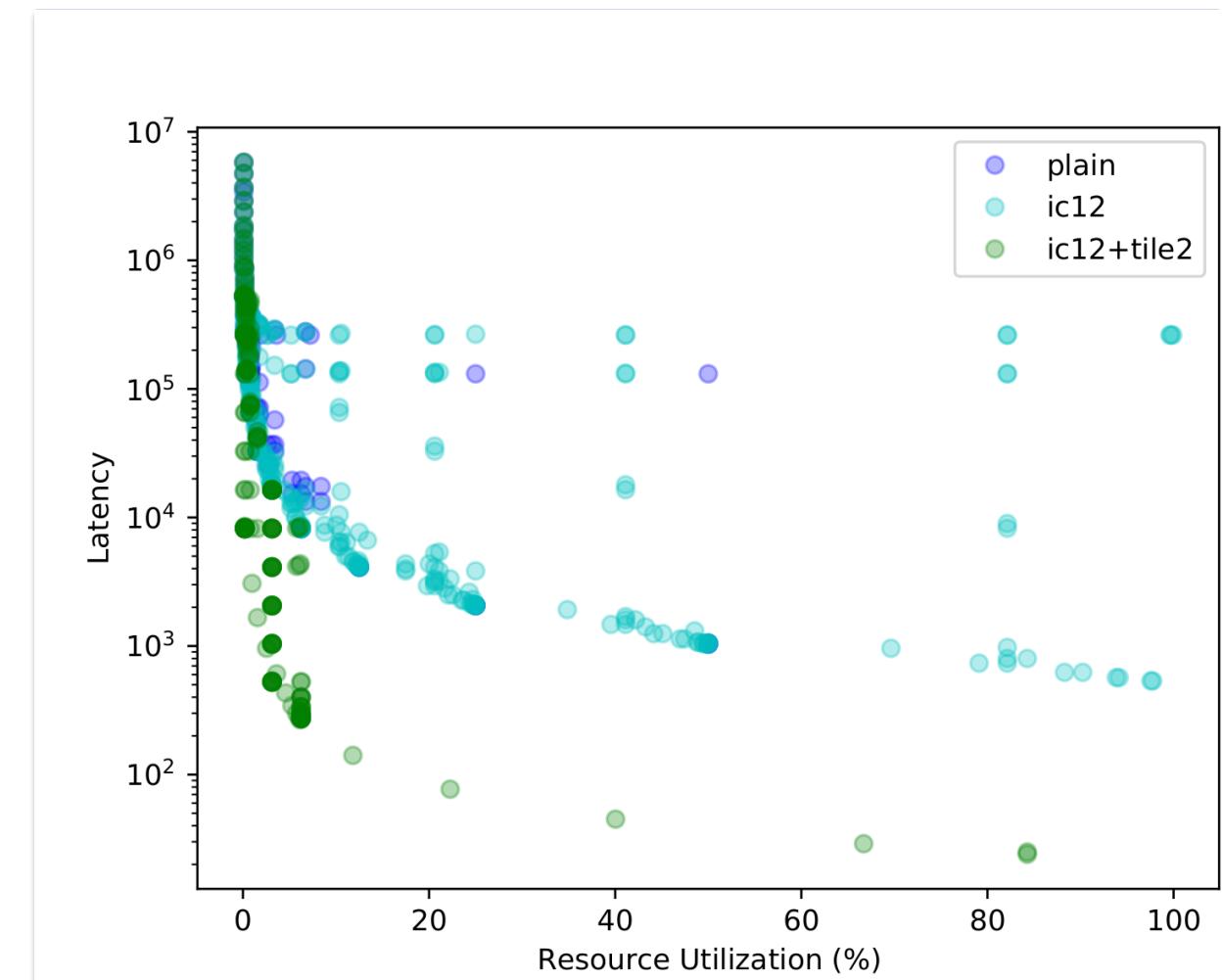
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)  
for22 (...)  
for11 (...)  
for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)  
for11 (...)  
for22 (...)  
for12 (...)
```

Design space of various code versions of 2D array addition.



("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:

$$c = a + b$$

Plain

```
for1 (...)  
for2 (...)
```

Interchange 1, 2

```
for2 (...)  
for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)  
for11 (...)  
for12 (...)
```

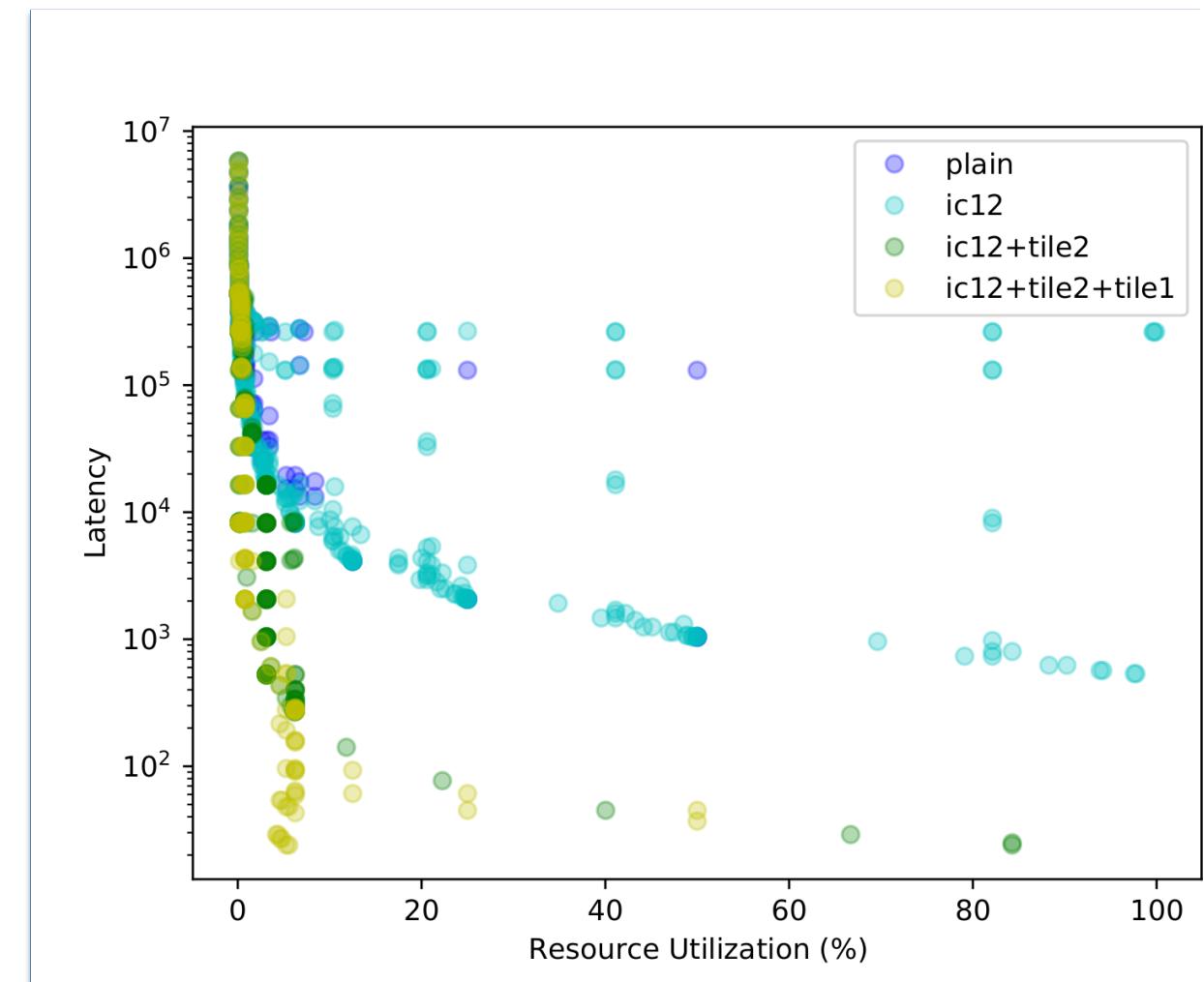
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)  
for22 (...)  
for11 (...)  
for12 (...)
```

Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)  
for11 (...)  
for22 (...)  
for12 (...)
```

Design space of various code versions of 2D array addition.

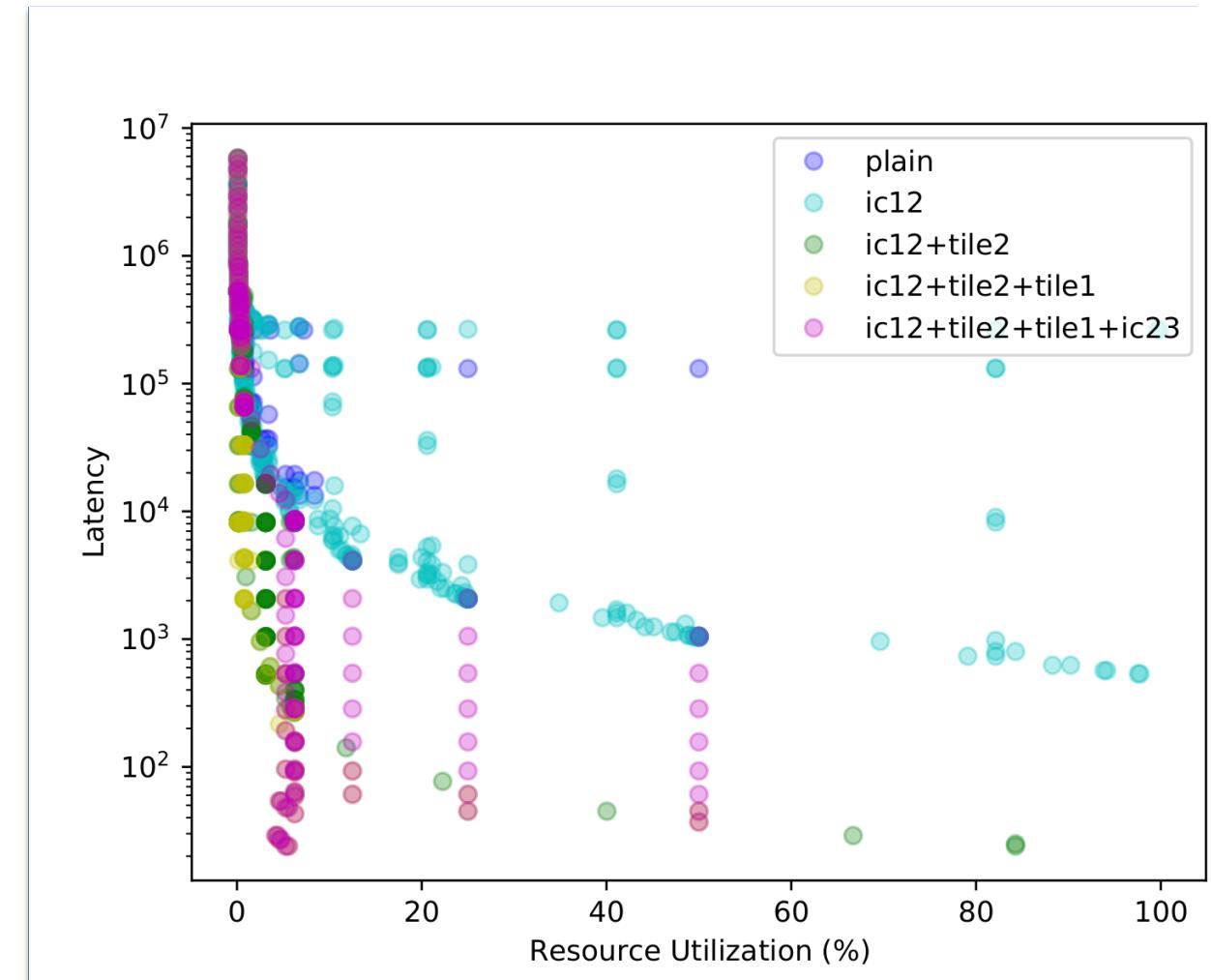


("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

2D Add:	
	$c = a + b$
Plain	for1 (...) for2 (...)
Interchange 1, 2	for2 (...) for1 (...)
Interchange 1, 2 Tile 2	for2 (...) for11 (...) for12 (...)
Interchange 1, 2 Tile 2 Tile 1	for21 (...) for22 (...) for11 (...) for12 (...)
Interchange 1, 2 Tile 2 Tile 1 Interchange 2, 3	for21 (...) for11 (...) for22 (...) for12 (...)

Design space of various code versions of 2D array addition.



Operator Implementation Selection

2D Add:

$$c = a + b$$

Plain

```
for1 (...)  
for2 (...)
```

Interchange 1, 2

```
for2 (...)  
for1 (...)
```

Interchange 1, 2
Tile 2

```
for2 (...)  
for11 (...)  
for12 (...)
```

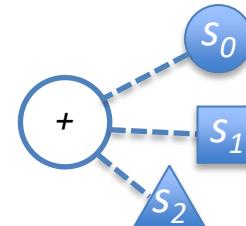
Interchange 1, 2
Tile 2
Tile 1

```
for21 (...)  
for22 (...)  
for11 (...)  
for12 (...)
```

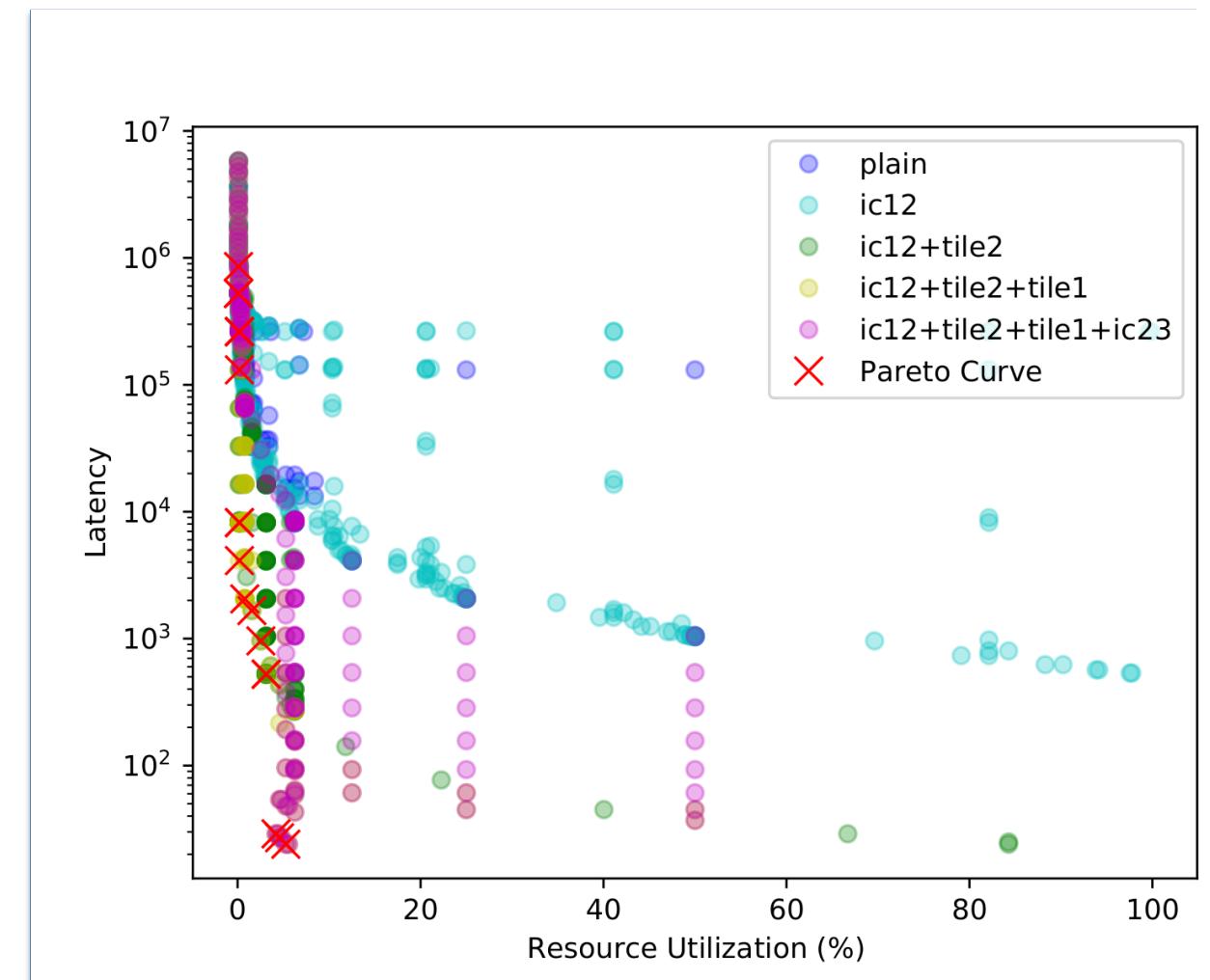
Interchange 1, 2
Tile 2
Tile 1
Interchange 2, 3

```
for21 (...)  
for11 (...)  
for22 (...)  
for12 (...)
```

Each high-level operations will have a list of candidate schedules, $schedules = [s_0, s_1, s_2, \dots]$



Design space of various code versions of 2D array addition.

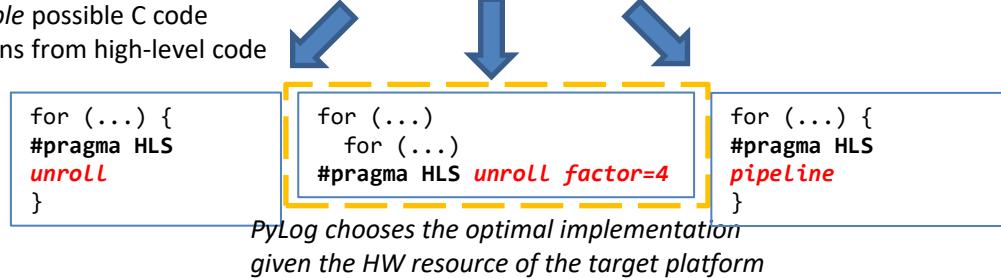


("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Operator Implementation Selection

```
1D Conv: y = map(Lambda a: a[-1]+a[0]+a[1], x[1:-1])
2D Conv: y = map(Lambda a: dot(a[-1:2, -1:2], w), img[1:-1, 1:-1])
```

Multiple possible C code versions from high-level code



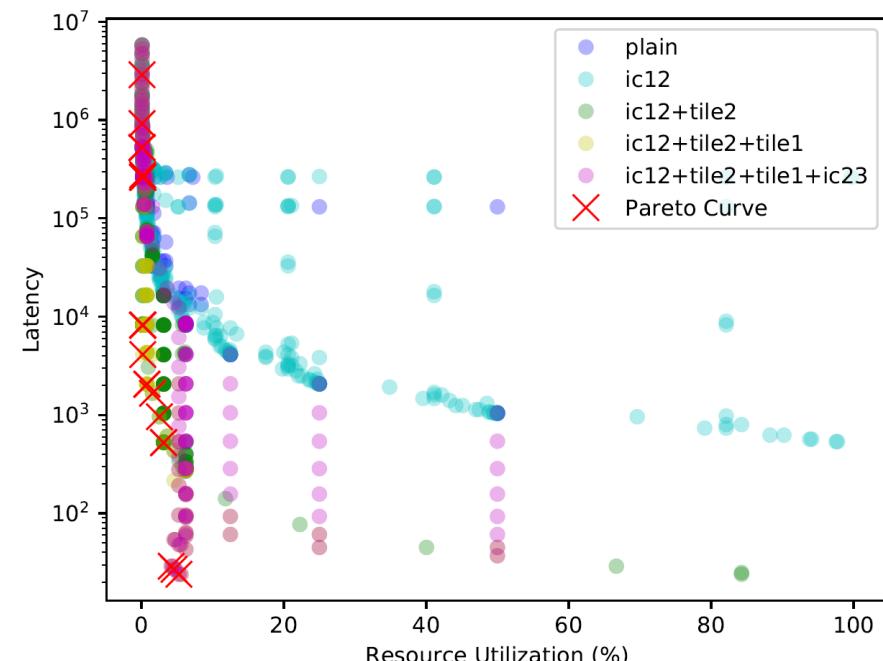
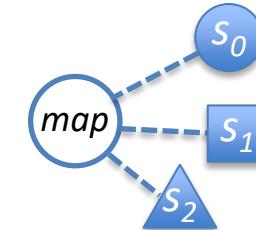
Applied to:

- N -dimensional array operations, e.g. $c = a + b$
- PyLog high-level operations, e.g. map, dot, reduce, etc.

PyLog IR:

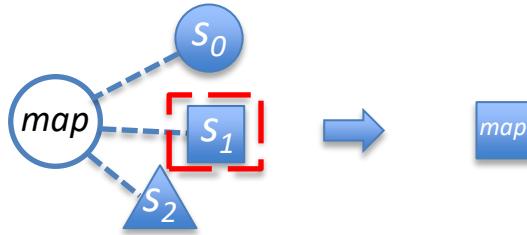
- Represent the transformations in IR systematically
- Represent performance and cost of each version

Each high-level operations will have a list of candidate schedules, $\text{schedule} = [s_0, s_1, s_2, \dots]$



Design space of various code versions of 2D array addition.
("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

Global Optimization - ILP Formulation

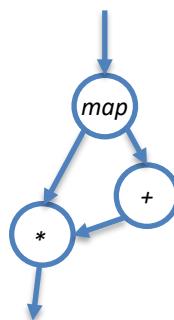


- The j^{th} implementation of i^{th} operator: x_{ij}
 $x_{ij} \in \{0,1\}$
- Each operator chooses one implementation
 $\forall i, \sum_j x_{ij} = 1$
- Latency and area (cost) of the j^{th} implementation of i^{th} operator: c_{ij} (l_{ij} or a_{ij})
- Total cost (assuming no branches):

$$\sum_{i,j} c_{ij} x_{ij}$$
- Total cost (with branches, generic CDFG):

$$\sum_{i,j} p_{ij} c_{ij} x_{ij}$$

where p_{ij} is the probability of operation x_{ij} being called.



Data Movement (Using CPU-FPGA interface as example)

- Where is the data / operator happening?

$$\sum_{j \text{ at FPGA}} x_{ij}$$

- Is there any data copy needed between operations p and q?

$$\sum_{j \text{ at FPGA}} x_{pj} - \sum_{j \text{ at FPGA}} x_{qj}$$

- Cost

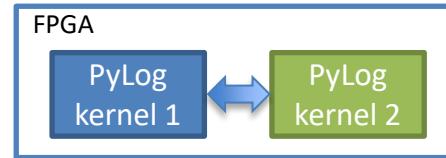
$$c_{F2C} \left(\sum_{j \text{ at FPGA}} x_{pj} - \sum_{j \text{ at FPGA}} x_{qj} \right)$$

- Total cost

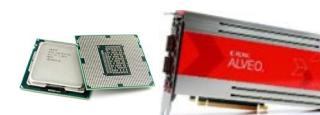
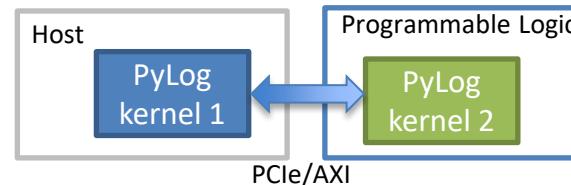
$$c_{F2C} \sum_{p \rightarrow q} \left(\sum_{j \text{ at FPGA}} x_{pj} - \sum_{j \text{ at FPGA}} x_{qj} \right)$$

Multiple Kernels

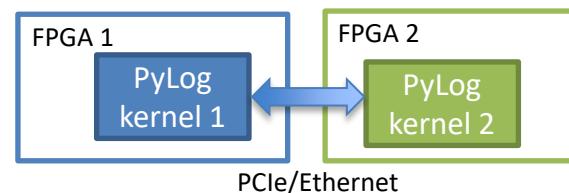
- Within same accelerator



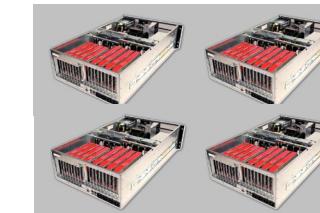
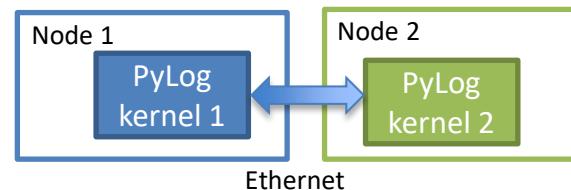
- Between host and accelerator



- Across accelerators



- Across nodes



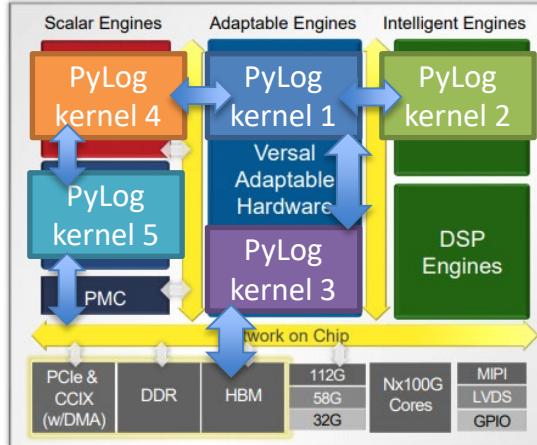
Multiple Kernels

```

@pylog
kernel 1 def acc1(a, b):
    ...
@pylog
kernel 2 def acc2(c, d):
    ...
host if __name__ == "__main__":
    a = np.array([1, 3, ...])
    b = np.array([8, 9, ...])
    d = np.array([8, 9, ...])
    combine(acc1, acc2)(a, b, d)

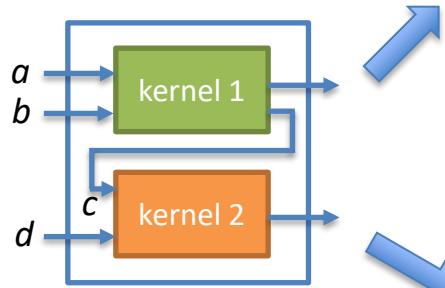
```

- New heterogeneous platforms



Xilinx Versal devices

- Returns a new function that combines kernels
- Connects inputs and outputs with kernels



Generic programmable logic: Vitis connectivity configuration

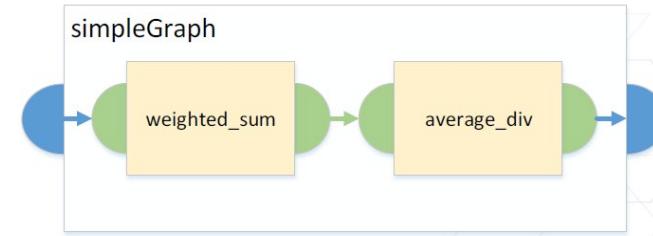
[connectivity]
#nk=<kernel_name>:<number>:<cu_name>.<cu_name>...
nk=vadd:3:vadd_X.vadd_Y.vadd_Z

#sp=<compute_unit_name>.<interface_name>:<bank_name>
sp=cnn_1.m_axi_gmem:DDR[0]

#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>
stream_connect=vadd_1.stream_out:vadd_2.stream_in

AI Engines: Dataflow model (on-going)

Connecting Kernels in graph.h



```

simpleGraph () {
    // Bind a function to each of the declared kernels
    k1 = kernel::create(weighted_sum);
    k2 = kernel::create(average_div);

    // create nets to connect kernels and IO ports
    connect<window<128>> net0 (in, k1.in[0]);
    connect<window<128>> net1 (k1.out[0], k2.in[0]);
    connect<window<128>> net2 (k2.out[0], out);
}

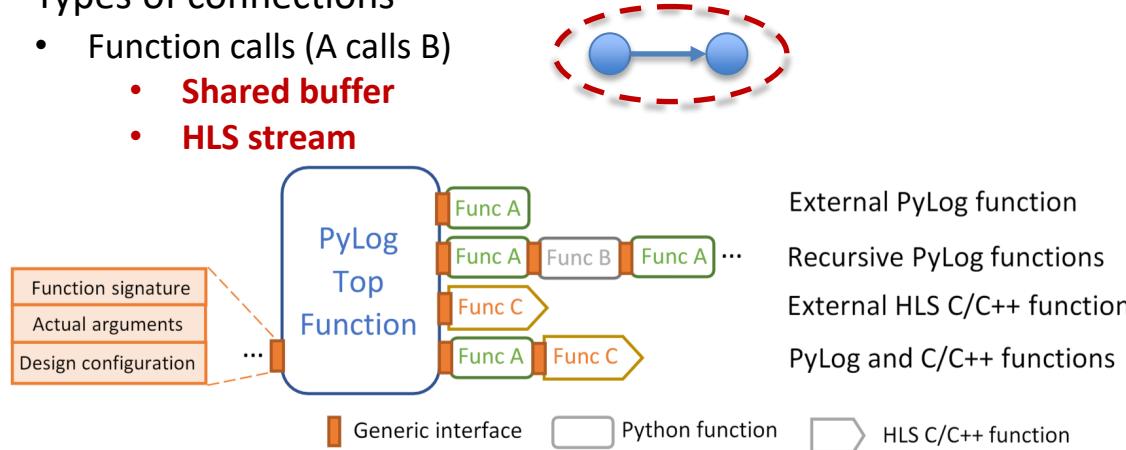
```

Multiple Kernels

1. Generic programmable logic: Inside IP (inside HLS)

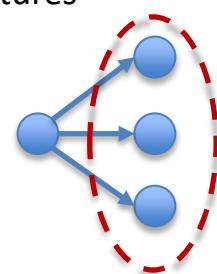
Types of connections

- Function calls (A calls B)
 - **Shared buffer**
 - **HLS stream**



- List of function calls connected with shared data structures
 - **Ping-pong buffers**
 - **FIFOs**

```
void top ( ... ) {
#pragma HLS dataflow
int A[1024];
#pragma HLS stream off variable=A depth=3
producer(A, B, ...); // producer writes A and B
middle(B, C, ...); // middle reads B and writes C
consumer(A, C, ...); // consumer reads A and C }
```



2. Generic programmable logic: IP level (system integration)
 - Shared DDR memory space
 - AXI-stream channel

[connectivity] **Creating Multiple Instances of a Kernel**
`#nk=<kernel name>:<number>:<cu_name>.<cu_name>...`
`nk=vadd:3:vadd_X.vadd_Y.vadd_Z`

Mapping Kernel Ports to Global Memory
`#sp=<compute_unit_name>.<interface_name>:<bank_name>`
`sp=cnn_1.m_axi_gmem:DDR[0]`

Specify streaming connections
`#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>`
`stream_connect=vadd_1.stream_out:vadd_2.stream_in`

3. Device level (CPU-accelerator)

- Shared virtual memory (AXI bus, memory mapped)
- Streaming port (AXI bus, streaming)
- Persistent memory?

KNN Example

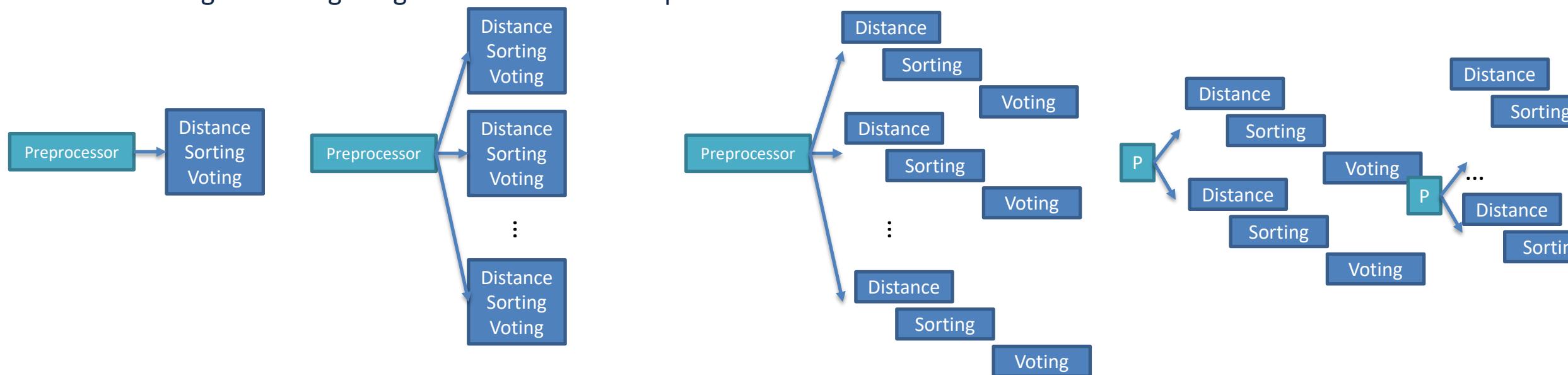
- Digit recognition KNN example

- Steps

- Preprocessing: Image -> feature vectors

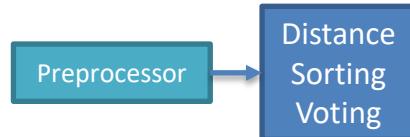


- Parallel for each test image:
 - Compare each test image against all training images
 - Sort distances in ascending order
 - Voting: k training images smallest distances predicts label

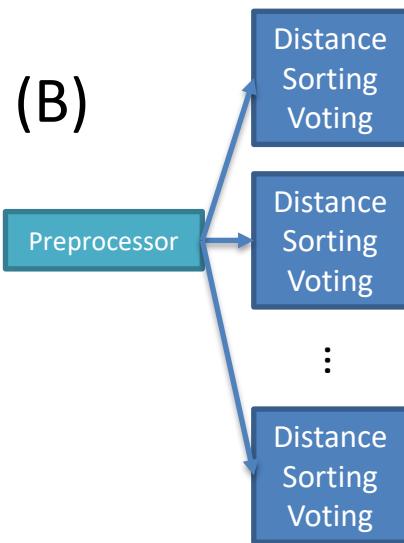


KNN Example

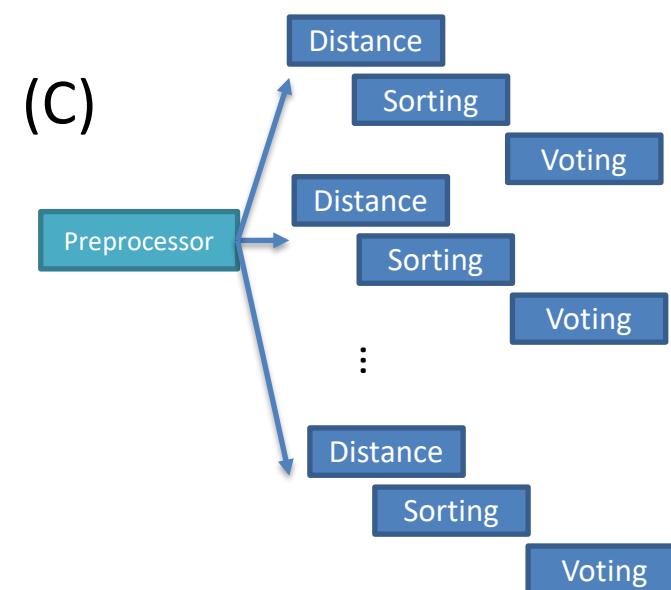
(A)



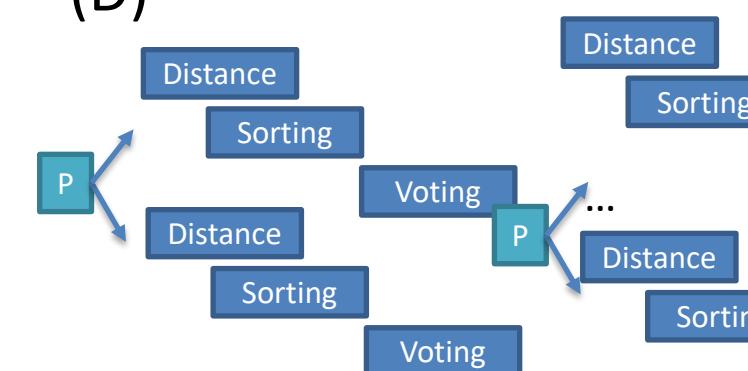
(B)



(C)



(D)



```
@pylog  
def prep(a, b):  
    ...  
@pylog  
def dist_sort_vote(c, d):  
    ...
```

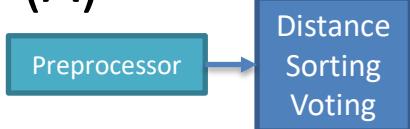
```
@pylog  
def prep(a, b):  
    ...  
@pylog  
@config(dup = N)  
def dist_sort_vote(c, d):  
    ...
```

```
@pylog  
def prep(a, b):  
    ...  
@config(dup = N)  
def dist_sort_vote(c, d):  
    @pylog  
    def dist(c, d):  
        ...  
  
@pylog  
def sort(c, d):  
    ...
```

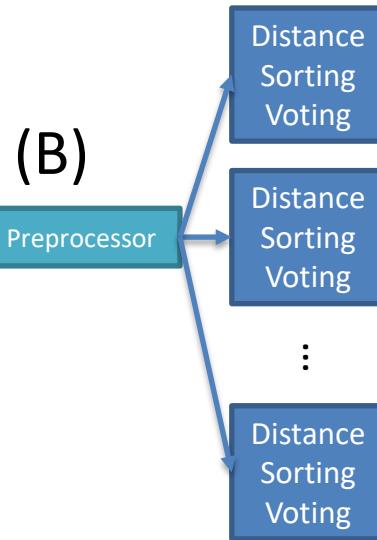
```
@pylog  
@config(dup = N)  
def prep(a, b):  
    ...  
@config(dup = N)  
def dist_sort_vote(c, d):  
    @pylog  
    def dist(c, d):  
        ...  
  
@pylog  
def sort(c, d):  
    ...
```

KNN Example

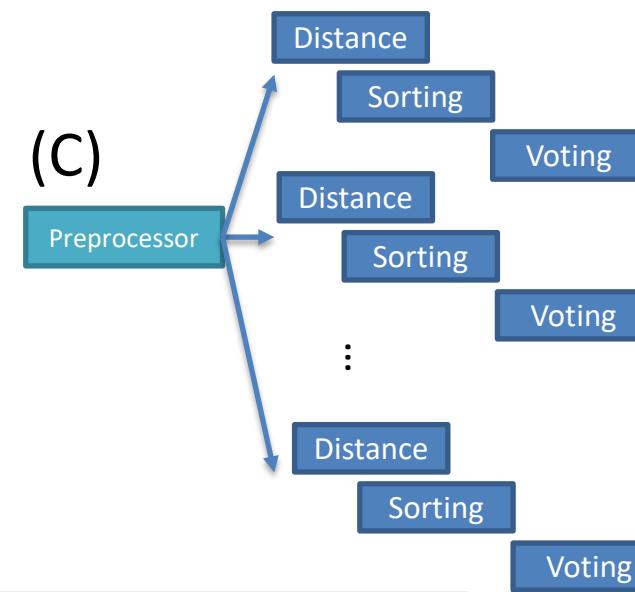
(A)



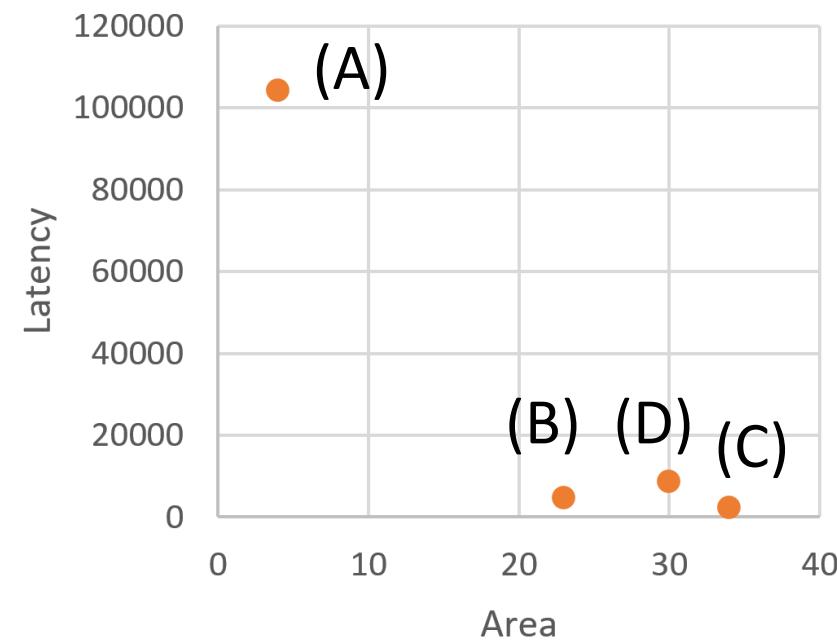
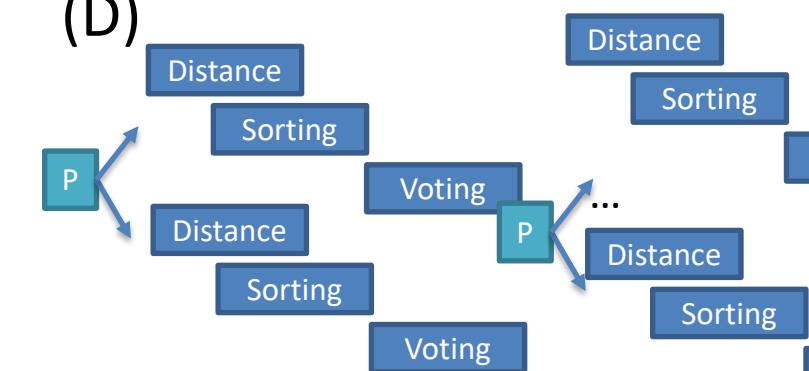
(B)



(C)



(D)



Verification in PyLog

PyLog Source Code

```
@pylog  
def accel(a, b, c, d, e):  
    d = (a + b) * c  
    e = map(lambda x, y: x+y), a,  
    d)
```

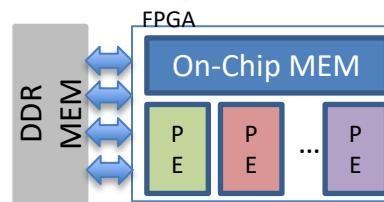
1

Optimized HLS C Code

```
void accel(int32* a, int32* b,  
          int32* c, int32* d, int32* e) {  
    for(int i = 0; i < 32; i++) {  
        for (ii...)  
            #pragma HLS ...  
    }  
    for(int j = 0; j < 32; j++) {  
        for (jj...)  
    }  
    ...  
}
```

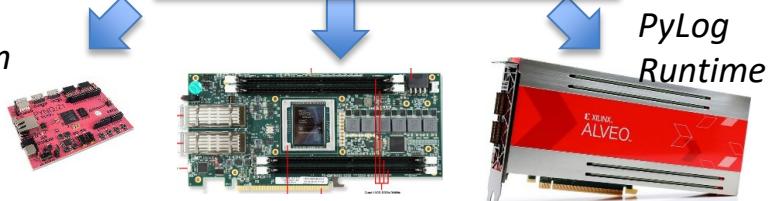
2

System Design



3

Target Platform Deployment



PyLog Runtime

4

UIUC ECE 527 - SoC Design (Fall 2020): Final Project by
Anjana S Kumar and **Christopher Baldwin**

1 PyLog code:

- functional correctness
- PyLog simulation: PySim



PyModel testing

2 **Generated HLS C code**

- Equivalence to PyLog code
- Test with PyLog input



PyLog verification flow:
@pylog(mode='verify')

3 RTL code

- Equivalence to HLS C code
- Vivado HLS RTL co-simulation



Vendor's simulation tools

4 On-board testing correctness

- PyLog runtime tests



PyLog runtime tests

▪ Input: randomly generated test vectors, with various data types, input lengths, corner cases

▪ Check:

- Generate and insert asserts into code
- Compare outputs from PyLog code and outputs from C code

Verification in PyLog

Anjana S Kumar, Christopher Baldwin

Steps

- Clean up input PyLog file
- Identifies input and output arguments
- Compiles PyLog generated C file to a shared library
- Generates test vectors
- Run PyLog code with PySim
- Run top C function (from Python)
- Compare outputs from C and Python



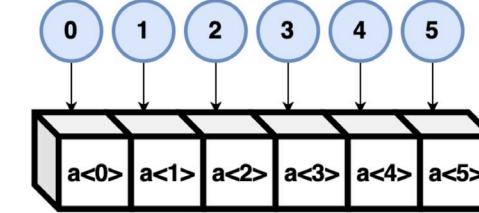
High-Level Descriptive Operators

All to help express spatial structures to the compiler

Parallelism Generation

gen.for

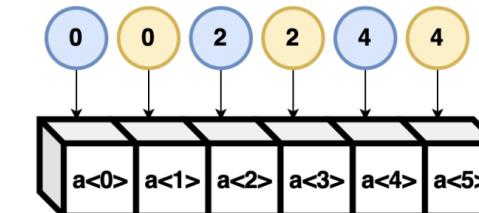
```
1 gen.for I in range(5):  
2     a<I> = I
```



Spatial Irregularity Generation

gen.if gen.elif gen.else

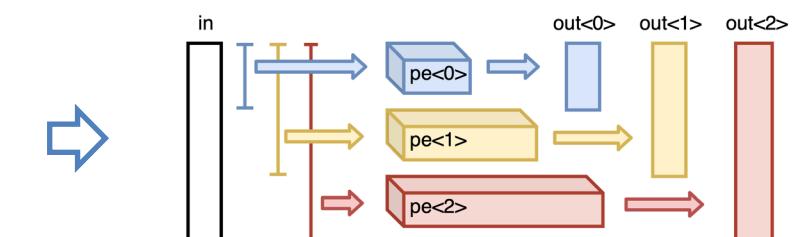
```
1 gen.for I in range(5):  
2     gen.if I % 2 == 0:  
3         a<I> = I  
4     gen.else:  
5         a<I> = I
```



Configurable Kernel Generation

.config() attribute

```
1 @config(batch)  
2 def pe(in,out):  
3     ...do work...  
4  
5 gen.for I in range(3):  
6     pe<I>(in,out<I>).config(I*100)
```



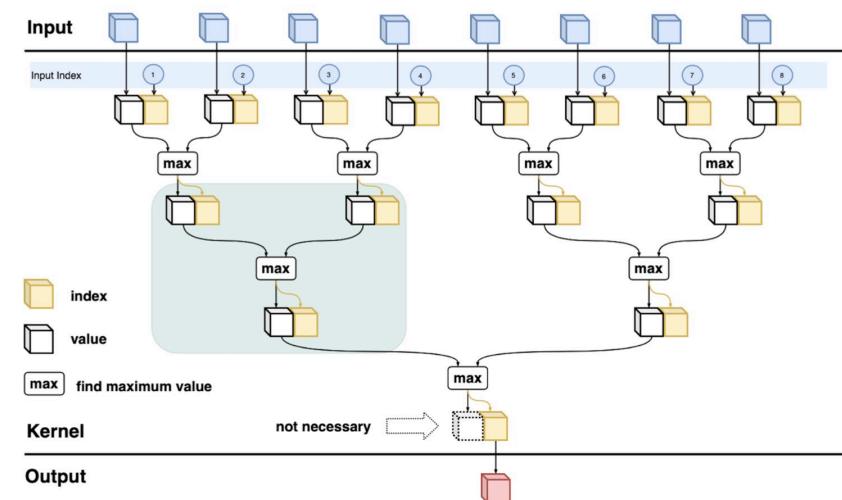
High-Level Descriptive Operators

```
1 @config(dict = [0:4, 1:2, 3:1])
2 def argmax(b) :
3
4     gen.for I in range(3):
5         gen.for J in range(dict[I]):
6             gen.if ( I == 2 ):
7                 ## loop body dedicated to the last loop ##
8                 gen.else:
9                     if ( b<I-1,J*2> > b<I-1,J*2+1> ):
10                         b<I,J> = b<I-1,J*2>
11                         index_b<I,J> = index_b<I-1,J>
12                 else:
13                     b<I,J> = b<I-1,J*2+1>
14                     index_b<I,J> = index_b<I-1,J+1>
15
16     return index_b<2,0>
```

Code 4. argmax Example.

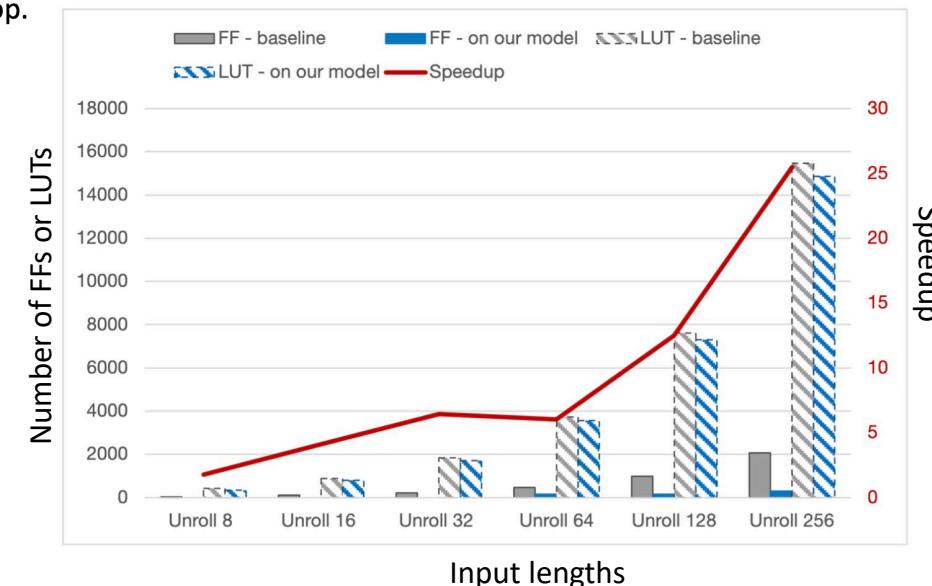


The implementation graph of the code using new syntax (unroll factor = 8)



```
int argmax(int input[SIZE]){
#pragma HLS array_partition variable=input complete
    int max_v = 0;
    int max_id = 0;
    for (int i=0; i<SIZE; i++){
#pragma HLS unroll
        if (input[i] > max_v){
            max_id = i;
            max_v = input[i];
        }
    }
    return max_id;
}
```

Code 6. Baseline Algorithm Argmax Using Regular for Loop.



Future Directions

Compilation

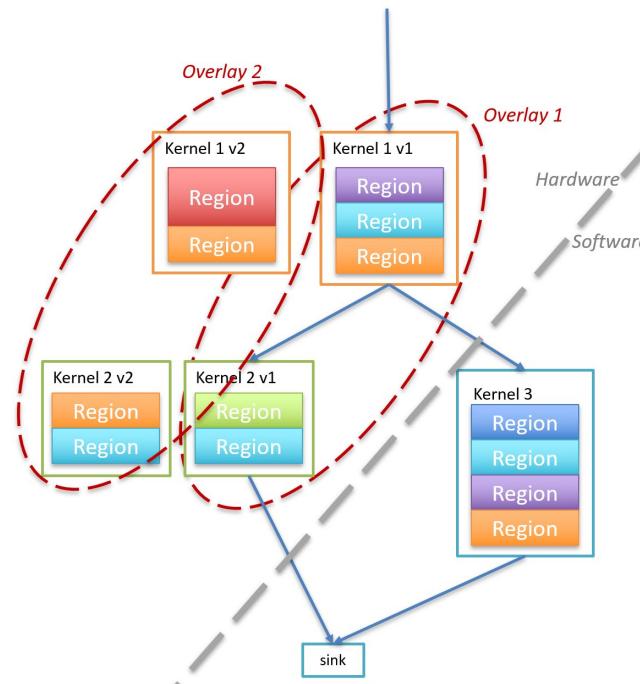
- Better support on Python/NumPy/Panda ops
- HW/SW partitioning and collaboration
- Fine-grained synthesis (op level)
- Compilation error handling – falling back to CPU
- GPU kernel generation
- Targeting new architectures like AI engines

Applications

- Deep learning
- Graph processing
- Data table operations
- Lightweight Cryptography

Use/Deployment Modes

- Better integration in the cloud/distributed environment
 - Refs: Dask for GPU, Numba for CPUs
- Closer integration with PYNQ environment
 - Utilize latest features provided by PYNQ
 - Pipeline templates, partial reconfiguration, etc.
 - Refs: Raspberry Pi flows for Arm CPUs



Reading Materials

- PyLog: An Algorithm-Centric FPGA Programming and Synthesis Flow. [[paper](#)]
- HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. [[paper](#)][[slides](#)]
- ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. [[paper](#)]
- Predictable Accelerator Design with Time-Sensitive Affine Types. [[paper](#)][[demo](#)]
- Chisel/FIRRTL Hardware Compiler Framework. [[webpage](#)]
- TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. [[webpage](#)]
- The Versatile Tensor Accelerator (VTA). [[webpage](#)]
- TACO: The Tensor Algebra Compiler. [[webpage](#)]

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



Lecture 7:

Machine Learning Compilers

Sitao Huang

sitaoh@uci.edu

February 22, 2022

(Ref: UIUC ECE 498 ICC: IoT and Cognitive Computing)



Tentative Schedule

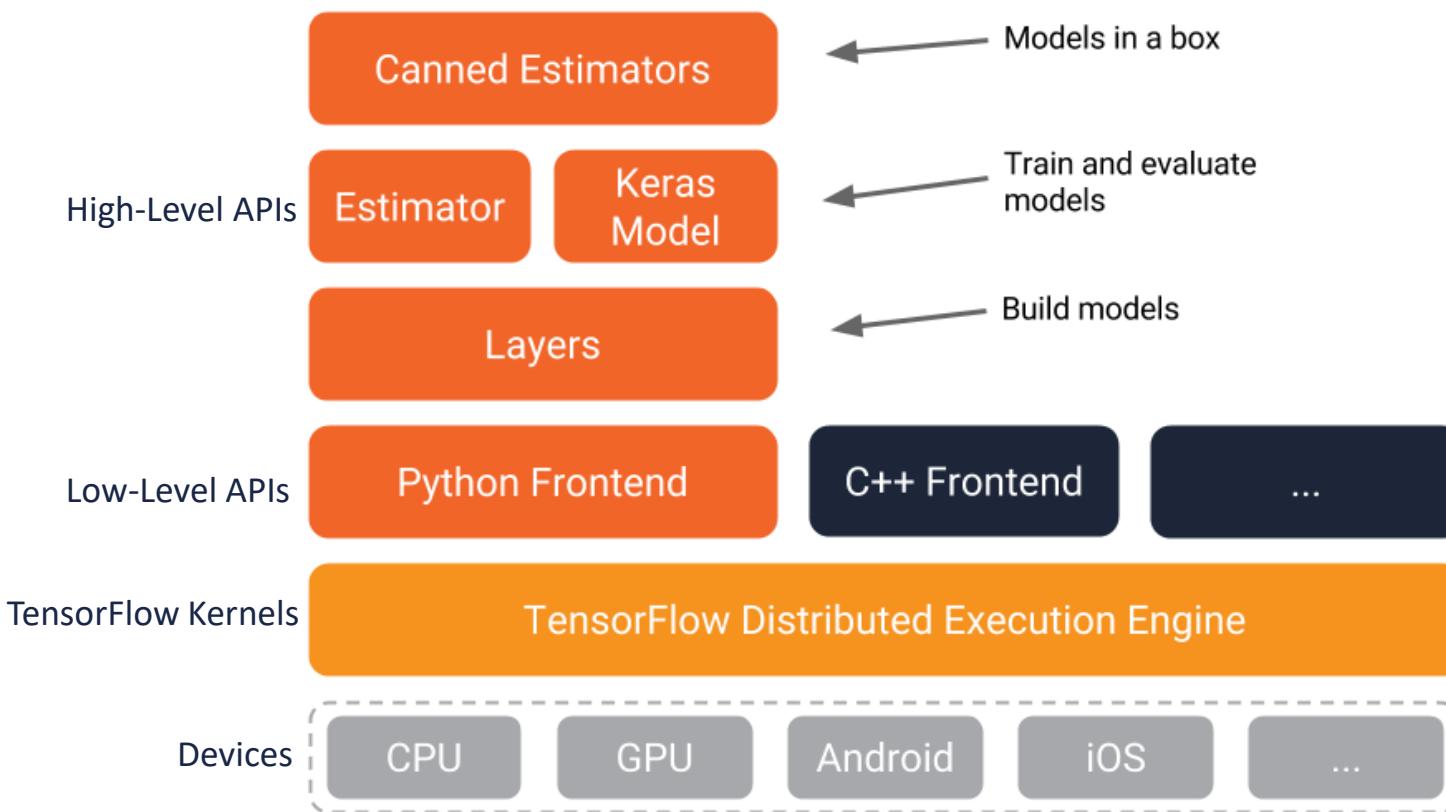
- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compilers for Accelerators
- **Week 8 (2/22, 2/24): Machine Learning Compilers**
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*



TensorFlow

TensorFlow

- TensorFlow is an open-source machine learning library for research and production.

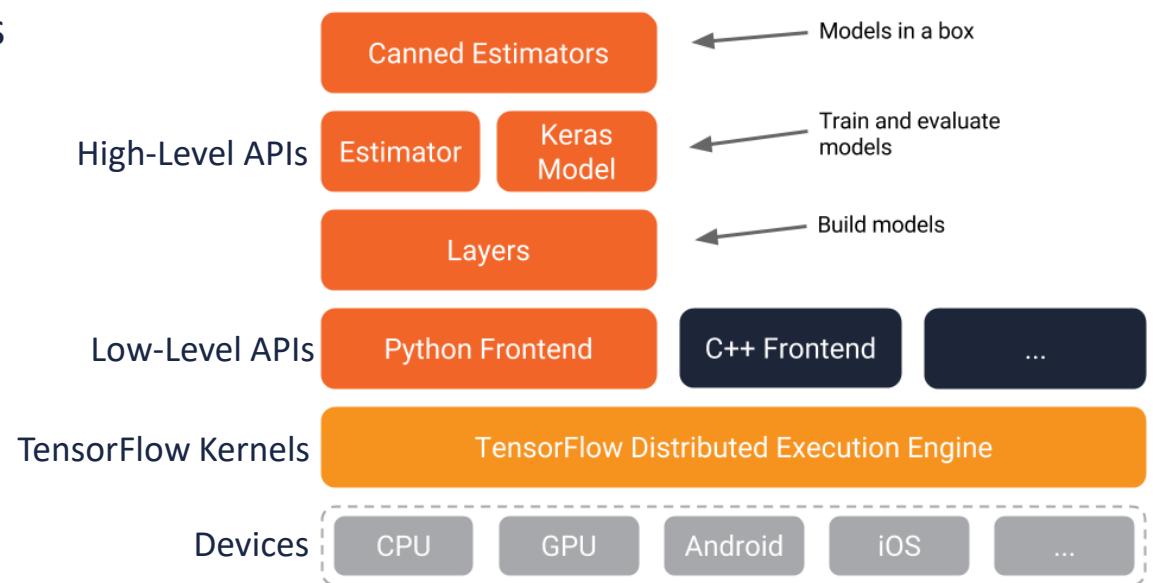


TensorFlow

- High-level APIs
 - [Keras](#), TensorFlow's high-level API for building and training deep learning models.
 - [Eager Execution](#), an API for writing TensorFlow code imperatively, without building graph.
 - [Importing Data](#), easy input pipelines to bring your data into your TensorFlow program.
 - [Estimators](#), a high-level API that provides fully-packaged models ready for large-scale training and production.
- Low-level APIs
 - [Tensors](#), the fundamental object in TensorFlow.
 - [Variables](#), represent shared, persistent state in program.
 - [Graphs](#), TensorFlow's representation of computations as dependencies between operations
 - [Sessions](#), TensorFlow's mechanism for running dataflow graphs across one or more local or remote devices.

If you are programming with the **low-level** TensorFlow API, graphs and sessions unit is essential.

If you are programming with a **high-level** TensorFlow API such as Estimators or Keras, the high-level API creates and manages graphs and sessions for you, but understanding graphs and sessions can still be helpful



Tensors

- Tensor
 - Generalization of vectors and matrices
 - Represented as n-dimensional arrays of base datatypes, `tf.Tensor`
- TensorFlow: a framework to define and run computations involving tensors
 - Build a graph of `tf.Tensor` objects
 - Detail how each tensor is computed based on the other available tensors
 - Run parts of this graph to achieve the desired results

Some types of Tensors:

- `tf.Variable`
- `tf.constant`
- `tf.placeholder`
- `tf.SparseTensor`

• tf.Variable

- A tf.Variable represents a tensor whose value can be changed by running ops on it.
- You add a variable to the graph by constructing an instance of the class Variable.

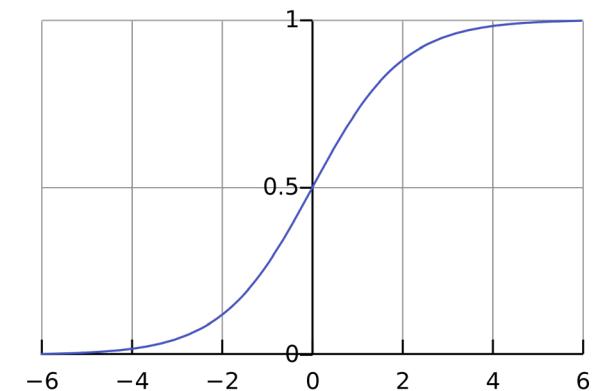
```
import tensorflow as tf

# Create a variable.
w = tf.Variable(<initial-value>, name=<optional-name>)

# Use the variable in the graph like any Tensor.
y = tf.matmul(w, ...another variable or tensor...)

# The overloaded operators are available too.
z = tf.sigmoid(w + y)

# Assign a new value to the variable with `assign()` or a related method.
w.assign(w + 1.0)
w.assign_add(1.0)
```



Sigmoid function

- `tf.constant`

```
tf.constant(  
    value,  
    dtype=None,  
    shape=None,  
    name='Const',  
    verify_shape=False  
)
```

- Creates a constant tensor.
 - The resulting tensor is populated with values of type `dtype`, as specified by arguments `value` and (optionally) `shape`

• **tf.placeholder**

```
tf.placeholder(  
    dtype,  
    shape=None,  
    name=None  
)
```

- Inserts a placeholder for a tensor that will be always fed.
- This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))  
y = tf.matmul(x, x)  
  
with tf.Session() as sess:  
    print(sess.run(y)) # ERROR: will fail because x was not fed.  
  
rand_array = np.random.rand(1024, 1024)  
print(sess.run(y, feed_dict={x: rand_array})) # Will succeed.
```

- **tf.SparseTensor**
 - Represents a sparse tensor.
 - TensorFlow represents a sparse tensor as three separate dense tensors: indices, values, and dense_shape.
 - indices: A 2-D int64 tensor of shape [N, ndims], which specifies the indices of the elements in the sparse tensor that contain N nonzero values (elements are zero-indexed), each is ndims. For example, indices=[[1,3], [2,4]] specifies that the elements with indexes of [1,3] and [2,4] have nonzero values.
 - values: A 1-D tensor of any type and shape [N], which supplies the values for each element in indices. For example, given indices=[[1,3], [2,4]], the parameter values=[18, 3.6] specifies that element [1,3] of the sparse tensor has a value of 18, and element [2,4] of the tensor has a value of 3.6.
 - dense_shape: A 1-D int64 tensor of shape [ndims], which specifies the dense_shape of the sparse tensor. Takes a list indicating the number of elements in each dimension. For example, dense_shape=[3,6] specifies a two-dimensional 3x6 tensor, dense_shape=[2,3,4] specifies a three-dimensional 2x3x4 tensor, and dense_shape=[9] specifies a one-dimensional tensor with 9 elements.

```
SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense_shape=[3, 4])
```

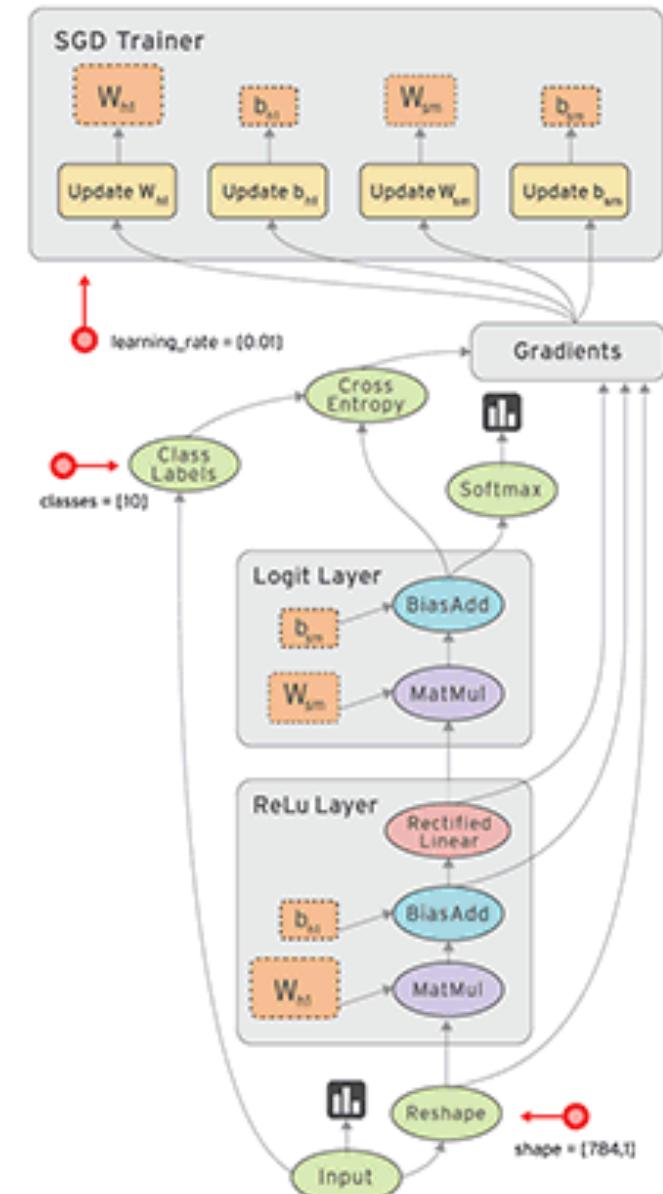
represents

```
[[1, 0, 0, 0]
 [0, 0, 2, 0]
 [0, 0, 0, 0]]
```

Dataflow Graphs

TensorFlow uses a **dataflow graph** (`tf.Graph`) to represent your computation in terms of the dependencies between individual operations.

- Low-level programming in TensorFlow:
 - first define the dataflow graph
 - then create a TensorFlow **session** to run parts of the graph across a set of local and remote devices
- Why Dataflow Graphs?
 - **Parallelism**: explicit data dependencies
 - **Distributed execution**: partition to multiple devices
 - **Compilation**: enable compiler optimization e.g. fusion
 - **Portability**: language-independent



Sessions

- TensorFlow uses the `tf.Session` class to represent a connection between the client program and the runtime.
 - provide access to devices in the local machine
 - provide access to remote devices using the distributed TensorFlow runtime
 - caches `tf.Graph` information
- Creating a `tf.Session`

```
# Create a default in-process session.
with tf.Session() as sess:
    # ...

# Create a remote session.
with tf.Session("grpc://example.org:2222"):
    # ...
```

- `tf.Session` owns physical resources (such as GPUs and network connections)
- used as a context manager (in a `with` block) that automatically closes the session when you exit the block.

Executing a graph in a tf.Session

The `tf.Session.run` method is the main mechanism for running a `tf.Operation` or evaluating a `tf.Tensor`. You can pass one or more `tf.Operation` or `tf.Tensor` objects to `tf.Session.run`, and TensorFlow will execute the operations that are needed to compute the result.

- Example:

```
x = tf.constant([[37.0, -23.0], [1.0, 4.0]])
w = tf.Variable(tf.random_uniform([2, 2]))
y = tf.matmul(x, w)
output = tf.nn.softmax(y)
init_op = w.initializer

with tf.Session() as sess:
    # Run the initializer on `w`.
    sess.run(init_op)

    # Evaluate `output`. `sess.run(output)` will return a NumPy array containing
    # the result of the computation.
    print(sess.run(output))

    # Evaluate `y` and `output`. Note that `y` will only be computed once, and its
    # result used both to return `y_val` and as an input to the `tf.nn.softmax()`
    # op. Both `y_val` and `output_val` will be NumPy arrays.
    y_val, output_val = sess.run([y, output])
```

Initializer

When you launch the graph, variables have to be explicitly initialized before you can run Ops that use their value. You can initialize a variable by running its initializer op, restoring the variable from a save file, or simply running an assign Op that assigns a value to the variable. In fact, the variable initializer op is just an assign Op that assigns the variable's initial value to the variable itself.

Nearest Neighbor Example

Load libraries, import data:

```
import numpy as np
import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

Construct dataflow graph:

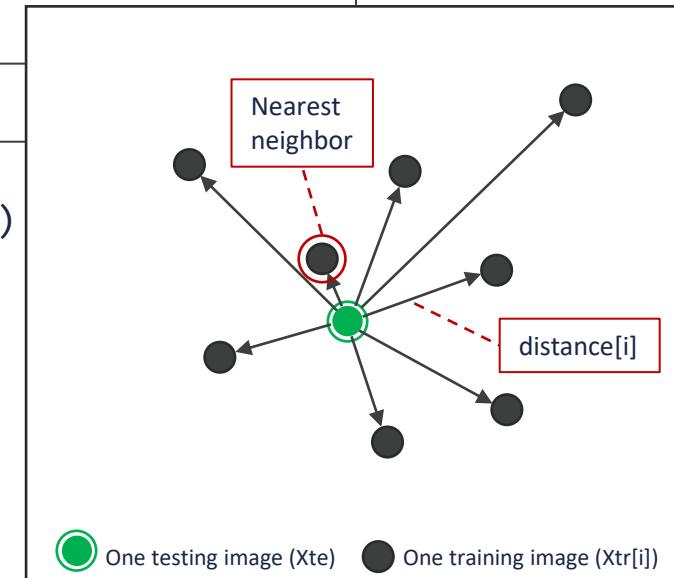
```
Training images (Xtr) and labels (Ytr) # In this example, we limit mnist data
Xtr, Ytr = mnist.train.next_batch(5000) #5000 for training (nn candidates)
Xte, Yte = mnist.test.next_batch(200) #200 for testing

Testing images (Xte) and labels (Yte) # tf Graph Input
xtr = tf.placeholder("float", [None, 784])
xte = tf.placeholder("float", [784]) # Each image has 28*28 = 784 pixels, used as feature vector

# Nearest Neighbor calculation using L1 Distance
# Calculate L1 Distance
distance = tf.reduce_sum(tf.abs(tf.add(xtr, tf.negative(xte)))), reduction_indices=1)
# Prediction: Get min distance index (Nearest neighbor)
pred = tf.argmin(distance, 0)

accuracy = 0.

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```



L1 distance:

$$\|x^{tr} - x^{te}\|_1 = \sum_i |x_i^{tr} - x_i^{te}|$$

Nearest Neighbor Example

Training:

```
with tf.Session() as sess:  
    sess.run(init)  
  
    # loop over test data  
    for i in range(len(Xte)):  
        # Get nearest neighbor  
        nn_index = sess.run(pred, feed_dict={xtr: Xtr, xte: Xte[i, :]})  
        # Get nearest neighbor class label and compare it to its true label  
        print "Test", i, "Prediction:", np.argmax(Ytr[nn_index]), \  
              "True Class:", np.argmax(Yte[i])  
        # Calculate accuracy  
        if np.argmax(Ytr[nn_index]) == np.argmax(Yte[i]):  
            accuracy += 1./len(Xte)  
    print "Done!"  
    print "Accuracy:", accuracy
```

Using GPUs

If a TensorFlow operation has both CPU and GPU implementations, the GPU devices will be given priority when the operation is assigned to a device*. For example, matmul has both CPU and GPU kernels. On a system with devices cpu:0 and gpu:0, gpu:0 will be selected to run matmul.

(*need to use GPU-enabled TensorFlow)

Using a single GPU in multi-GPU system

```
# Creates a graph.
with tf.device('/device:GPU:2'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
    c = tf.matmul(a, b)
# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(c))
```

Using multiple GPUs

```
# Creates a graph.
c = []
for d in ['/device:GPU:2', '/device:GPU:3']:
    with tf.device(d):
        a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
        b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
        c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
    sum = tf.add_n(c)
# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(sum))
```

Device mapping:
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: Tesla K20m, pci bus
id: 0000:02:00.0
/job:localhost/replica:0/task:0/device:GPU:1 -> device: 1, name: Tesla K20m, pci bus
id: 0000:03:00.0
/job:localhost/replica:0/task:0/device:GPU:2 -> device: 2, name: Tesla K20m, pci bus
id: 0000:83:00.0
/job:localhost/replica:0/task:0/device:GPU:3 -> device: 3, name: Tesla K20m, pci bus
id: 0000:84:00.0
Const_3: /job:localhost/replica:0/task:0/device:GPU:3
Const_2: /job:localhost/replica:0/task:0/device:GPU:3
MatMul_1: /job:localhost/replica:0/task:0/device:GPU:3
Const_1: /job:localhost/replica:0/task:0/device:GPU:2
Const: /job:localhost/replica:0/task:0/device:GPU:2
MatMul: /job:localhost/replica:0/task:0/device:GPU:2
AddN: /job:localhost/replica:0/task:0/cpu:0
[[44. 56.]
 [98. 128.]]

Cluster Specification

The cluster specification dictionary `tf.train.ClusterSpec` maps job names to lists of network addresses. For example,

```
tf.train.ClusterSpec({"local": ["localhost:2222",  
                               "localhost:2223"]}) → /job:local/task:0  
                                /job:local/task:1
```

```
tf.train.ClusterSpec({  
    "worker": [  
        "worker0.example.com:2222",  
        "worker1.example.com:2222",  
        "worker2.example.com:2222"  
    ],  
    "ps": [  
        "ps0.example.com:2222",  
        "ps1.example.com:2222"  
    ]}) → /job:worker/task:0  
                    /job:worker/task:1  
                    /job:worker/task:2  
                    /job:ps/task:0  
                    /job:ps/task:1
```

Placing operations on different devices

- A **device specification** has the following form:

```
/job:<JOB_NAME>/task:<TASK_INDEX>/device:<DEVICE_TYPE>:<DEVICE_INDEX>
```

```
# Operations created outside either context will run on the "best possible"
# device. For example, if you have a GPU and a CPU available, and the operation
# has a GPU implementation, TensorFlow will choose the GPU.
weights = tf.random_normal(...)

with tf.device("/device:CPU:0"):
    # Operations created in this context will be pinned to the CPU.
    img = tf.decode_jpeg(tf.read_file("img.jpg"))

with tf.device("/device:GPU:0"):
    # Operations created in this context will be pinned to the GPU.
    result = tf.matmul(weights, img)
```

If you are deploying TensorFlow in a typical distributed configuration, you might specify the job name and task ID to place variables on a task in the parameter server job ("`/job:ps`"), and the other operations on task in the worker job ("`/job:worker`"):

```
with tf.device("/job:ps/task:0"):
    weights_1 = tf.Variable(tf.truncated_normal([784, 100]))
    biases_1 = tf.Variable(tf.zeros([100]))

with tf.device("/job:ps/task:1"):
    weights_2 = tf.Variable(tf.truncated_normal([100, 10]))
    biases_2 = tf.Variable(tf.zeros([10]))

with tf.device("/job:worker"):
    layer_1 = tf.matmul(train_batch, weights_1) + biases_1
    layer_2 = tf.matmul(train_batch, weights_2) + biases_2
```

TensorFlow Computation Graphs

TensorBoard

EVENTS IMAGES GRAPH HISTOGRAMS

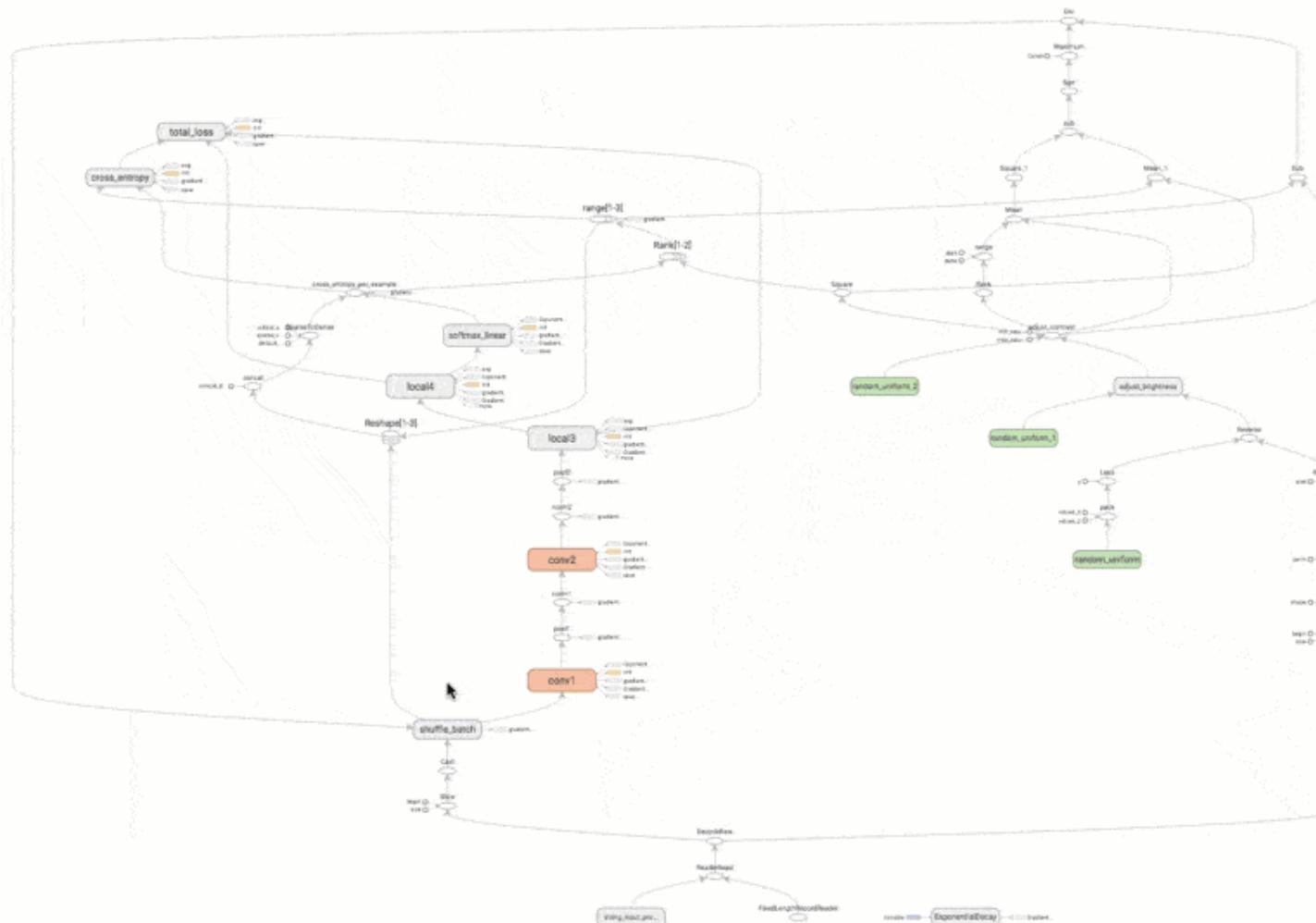
 Fit to screen

Run cifar-train

Upload Choose File

Color **Structure**

Main Graph



Auxiliary nodes

Graph (* = expandable)

Namespace*

OpNode

 Unconnected series*

Connected

Constant

Summary

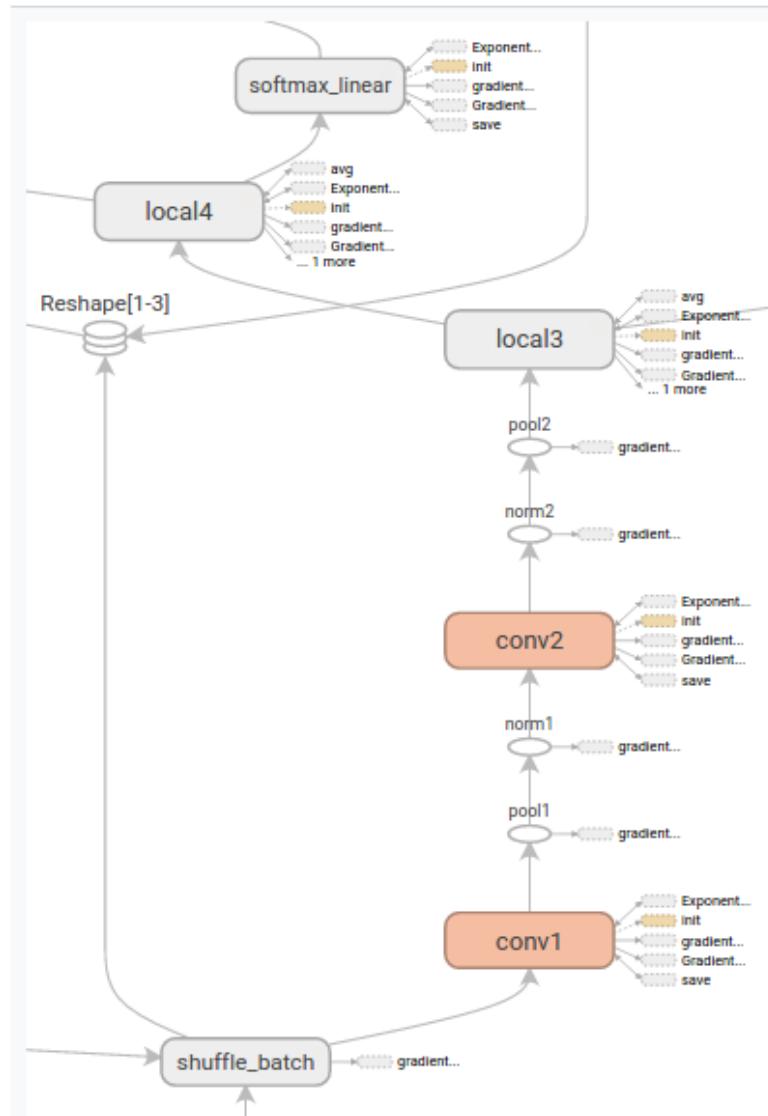
→ Dataflow edge

Control dependency

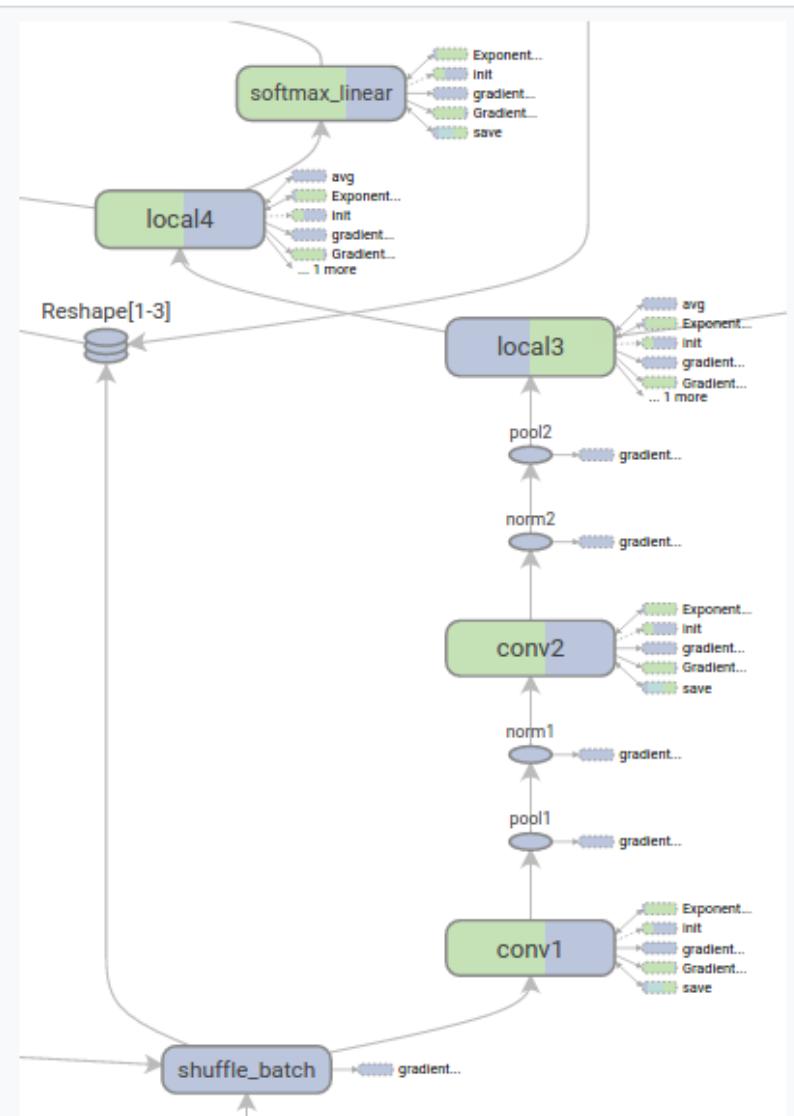
Reference edge

TensorFlow Computation Graphs

(Visualized in TensorBoard)



Structure view: The gray nodes have unique structure. The orange conv1 and conv2 nodes have the same structure, and analogously for nodes with other colors.



Device view: Name scopes are colored proportionally to the fraction of devices of the operation nodes inside them. Here, purple means GPU and the green is CPU.



TensorFlow Lite

TensorFlow Lite

A set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices.

Main Components

- [TensorFlow Lite interpreter](#): runs specially optimized models on many different hardware types, including mobile phones, embedded Linux devices, and microcontrollers.
- [TensorFlow Lite converter](#): converts TensorFlow models into an efficient form for use by the interpreter, and it can introduce *optimizations* to improve binary size and performance.

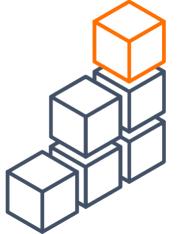
Key Features

- *Interpreter* tuned for on-device ML.
- Diverse platform support, covering [Android](#) and [iOS](#) devices, embedded Linux, and microcontrollers.
- APIs for multiple languages including Java, Swift, Objective-C, C++, and Python.
- High performance, with [hardware acceleration](#) on supported devices, device-optimized kernels, and [pre-fused activations and biases](#).
- Model optimization tools, including [quantization](#), that can reduce size and increase performance of models without sacrificing accuracy.
- Efficient model format, using a [FlatBuffer](#) that is optimized for small size and portability.
- [Pre-trained models](#) for common machine learning tasks that can be customized to your application.
- [Samples and tutorials](#) that show you how to deploy machine learning models on supported platforms.

Limitations

- Supports a limited subset of TensorFlow operators. You can use [TensorFlow Select](#) to include TensorFlow operations not yet supported.
- Currently no support for on-device training

TensorFlow Lite Development Flow



Pick a model

Bring your own TensorFlow model, find a model online, or pick a model from [Pre-trained models](#) to drop in or retrain.



Convert the model

If you're using a custom model, use the [TensorFlow Lite converter](#) and a few lines of Python to convert it to the TensorFlow Lite format.



Deploy to your device

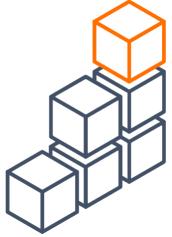
Run your model on-device with the [TensorFlow Lite interpreter](#), with APIs in many languages.



Optimize your model

Use [Model Optimization Toolkit](#) to reduce your model's size and increase its efficiency with minimal impact on accuracy.

TensorFlow Lite Development Flow



Pick a model

Bring your own TensorFlow model, find a model online, or pick a model from [pre-trained models](#) to drop in or retrain.

- A TensorFlow model is a data structure that contains the logic and knowledge of a machine learning network trained to solve a particular problem.
- To use a model with TensorFlow Lite, you must start with a regular TensorFlow model, and then [convert the model](#). You cannot create or train a model using TensorFlow Lite.

Options

- Use a pre-trained TensorFlow / TensorFlow Lite model (list of [pre-trained models](#))
- Re-train a model (transfer learning) to your applications
- Train a custom model (covered in last lecture)

TensorFlow Lite Development Flow



Convert the model

If you're using a custom model, use the [TensorFlow Lite converter](#) and a few lines of Python to convert it to the TensorFlow Lite format.

Convert a TensorFlow `SavedModel` into the TensorFlow Lite format:

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_model)
```

It can also introduce optimizations, which are covered in later slides.

The converter can also be used from the [command line](#), but the Python API is recommended.

Input types for the converter:

- For TensorFlow 1.x models: [SavedModel directories](#), Frozen GraphDef (models generated by [freeze_graph.py](#)), [Keras](#) HDF5 models, Models taken from a `tf.Session`.
- For TensorFlow 2.x models: [SavedModel directories](#), [tf.keras models](#), [Concrete functions](#)

TensorFlow Lite Development Flow



Deploy to your device

Run your model on-device with the [TensorFlow Lite interpreter](#), with APIs in many languages.

TensorFlow Lite Interpreter

The [TensorFlow Lite interpreter](#) is a library that takes a model file, executes the operations it defines on input data, and provides access to the output. It provides a simple API for running TensorFlow Lite models from Java, Swift, Objective-C, C++, and Python.

GPU Acceleration and Delegates

The [GPU Delegate](#) allows the interpreter to run appropriate operations on the device's GPU.

An example of GPU Delegate being used from Java:

```
GpuDelegate delegate = new GpuDelegate();
Interpreter.Options options = (new Interpreter.Options()).addDelegate(delegate);
Interpreter interpreter = new Interpreter(tensorflow_lite_model_file, options);
```

To add support for new hardware accelerators you can [define your own delegate](#).

Platforms

- [Android](#) and [iOS](#)
- [Linux](#)
- Microcontrollers: Use [TensorFlow Lite for Microcontrollers](#)

TensorFlow Lite Development Flow



Optimize your model

Use [Model Optimization Toolkit](#) to reduce your model's size and increase its efficiency with minimal impact on accuracy.

Performance

The goal of model optimization is to reach the ideal balance of performance, model size, and accuracy on a given device. [Performance best practices](#) can help guide you through this process.

Quantization

TensorFlow Lite supports reducing precision of values from full floating point to half-precision floats (float16) or 8-bit integers: [post-training quantization](#). An example of quantizing: a SavedModel:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tfLite_quantized_model = converter.convert()
open("converted_model.tflite", "wb").write(tfLite_quantized_model)
```

Converts only the weights
from floating point to integer

Can also force all inputs and outputs to be quantized as well:

```
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8 # or tf.uint8
converter.inference_output_type = tf.int8 # or tf.uint8
```

Other Optimization

Check out [Model Optimization](#) for more information.

TensorFlow Lite for Microcontrollers

Microcontrollers

- A small computer on a single integrated circuit (IC) chip, contains CPU(s), memory and I/O peripherals
- Small memory (a few kilobytes), low power (a few milliwatts or even less than a milliwatt)
Ref: Arm Cortex-A9 CPU: 500mW
- Executes instructions directly (bare metal, no OS) or only a tailored minimum OS (no complete Linux)

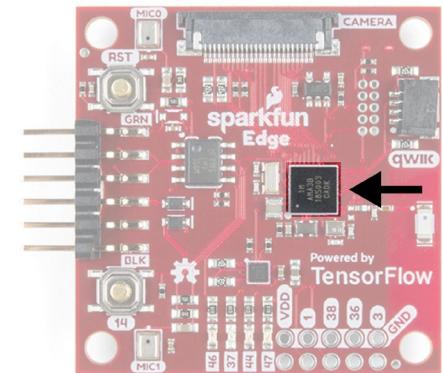
(Note: Raspberry Pi has more powerful hardware and embedded Linux, so we don't consider it as microcontroller here. Raspberry Pi can support standard TensorFlow Lite)

Why ML on Microcontrollers is Important?

- Microcontrollers are embedded in many devices to provide basic computation
- ML on microcontroller can boost the intelligence of billions of devices used in our lives
- Preserve privacy, no data leaves the device

TensorFlow Lite for Microcontrollers

- A special version of TensorFlow Lite designed to run ML models on *microcontrollers* and other devices with *only few kilobytes of memory*
- Doesn't require operating system support, any standard C or C++ libraries, or dynamic memory allocation
- List of [supported microcontrollers](#)



The SparkFun Edge board
(source: sparkfun.com)



Arduino Nano 33 BLE Sense board
(source: arduino.cc)

Run Inference on *Microcontrollers*

Let's look at a very basic "Hello World" example that loads model, prepares input, checks input shape, runs inference, and finally checks output.

1. Include the headers

```
// library headers
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
// model headers (contains g_model, a C array of weights)
#include "tensorflow/lite/micro/examples/hello_world/model.h"
// unit test framework headers
#include "tensorflow/lite/micro/testing/micro_test.h"
```

2. Set up logging

```
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* error_reporter = &micro_error_reporter;
```

3. Load a model

```
const tflite::Model* model = ::tflite::GetModel(g_model);
if (model->version() != TFLITE_SCHEMA_VERSION) {
    TF_LITE_REPORT_ERROR(error_reporter,
        "Model provided is schema version %d not equal "
        "to supported version %d.\n",
        model->version(), TFLITE_SCHEMA_VERSION);
}
```

Run Inference on *Microcontrollers*

4. Instantiate operations resolver (used by interpreter to access the operations used by the model)

```
tflite::AllOpsResolver resolver;
```

5. Allocate memory (for input, output and intermediate arrays)

```
const int tensor_arena_size = 2 * 1024;  
uint8_t tensor_arena[tensor_arena_size];
```

6. Instantiate interpreter

```
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,  
                                     tensor_arena_size, error_reporter);
```

7. Allocate tensors

```
interpreter.AllocateTensors();
```

8. Validate input shape

```
// Obtain a pointer to the model's input tensor  
TfLiteTensor* input = interpreter.input(0);  
  
// Make sure the input has the properties we expect  
TF_LITE_MICRO_EXPECT_NE(nullptr, input);  
TF_LITE_MICRO_EXPECT_EQ(2, input->dims->size);  
// The value of each element gives the length of the corresponding tensor.  
// We should expect two single element tensors (one is contained within the  
// other).  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);  
// The input is a 32 bit floating point value  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, input->type);
```

Run Inference on *Microcontrollers*

9. Provide an input value

```
input->data.f[0] = 0.;
```

10. Run the model

```
TfLiteStatus invoke_status = interpreter.Invoke();
if (invoke_status != kTfLiteOk) {
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed\n");
}
```

11. Obtain the output

```
TfLiteTensor* output = interpreter.output(0);
TF_LITE_MICRO_EXPECT_EQ(2, output->dims->size);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, output->type);

// Obtain the output value from the tensor
float value = output->data.f[0];
// Check that the output value is within 0.05 of the expected value
TF_LITE_MICRO_EXPECT_NEAR(0., value, 0.05);
```

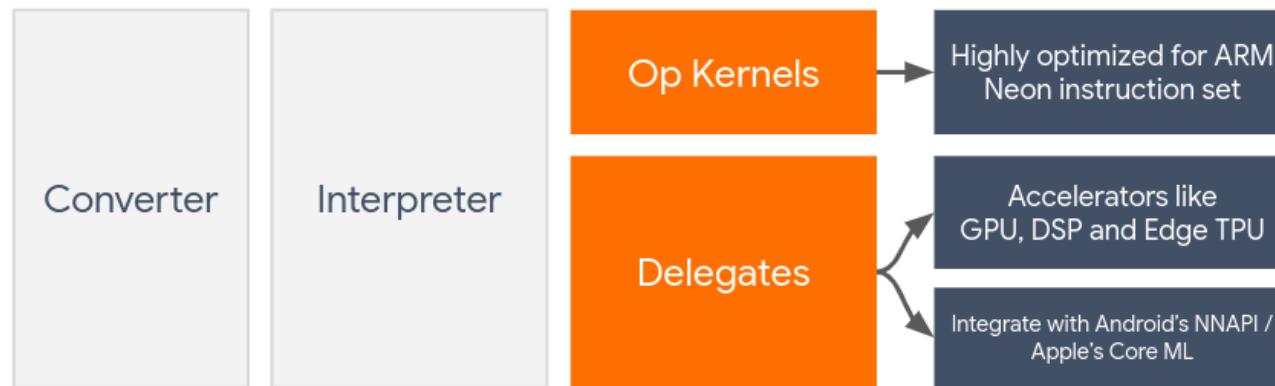
Complete code is here: [hello world test.cc](#).

TensorFlow Lite Delegates

- **Delegates** enable hardware acceleration of TensorFlow Lite models (part or whole) by leveraging on-device accelerators such as GPU, DSP, and Edge TPU.
- TensorFlow Lite default backend: CPU kernels optimized for [ARM Neon](#) instruction set.
- Most modern cell phones or embedded platforms contain specialized processors, each programmed with special APIs.
- TensorFlow Lite's Delegate API acts as a bridge between the TFLite runtime and these special lower-level APIs.

Delegate Examples

- **GPU delegate**: used on both Android and iOS, optimized to run 32-bit and 16-bit floating-point based models on GPU
- **NNAPI delegate** (Android devices with GPU, DSP, and NPU) and Hexagon delegate (Android devices with Hexagon DSP)
- **Core ML delegate** for Neural Engines in newer iPhones and iPads (A12 SoC or higher), accelerates inference for 32-bit or 16-bit floating-point models



Model Type	GPU	NNAPI	Hexagon	Core ML
Floating-point (32 bit)	Yes	Yes	No	Yes
Post-training float16 quantization	Yes	No	No	Yes
Post-training dynamic range quantization	Yes	Yes	No	No
Post-training integer quantization	Yes	Yes	Yes	No
Quantization-aware training	Yes	Yes	Yes	No

TensorFlow Lite Delegates – Custom Delegates

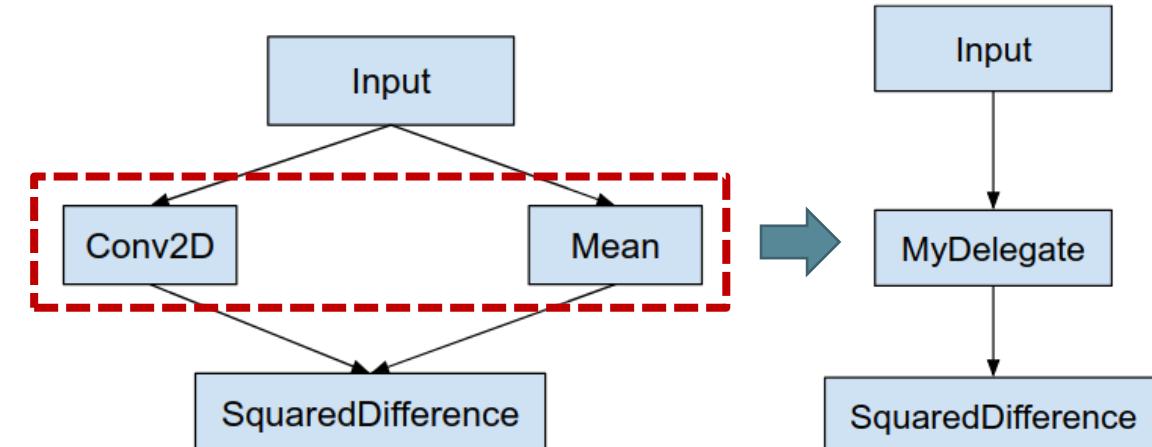
In some cases, you may need to create custom delegate:

- Integrate a ***new ML engine*** not supported by any existing delegate
- Have a ***custom hardware accelerator*** that improves runtime for known scenarios
- Developing CPU ***optimizations*** (such as operator fusing) that can speed up certain models

TensorFlow Lite Graph Transformation Rules (when applying

custom delegates):

- Specific operations that could be handled by the delegate are put into a partition while still satisfying the original computing workflow dependencies among operations.
(supported operations are put into a partition)
- Each to-be-delegated partition only has input and output nodes that are not handled by the delegate.
(the sizes of these partitions are maximized/aggregated whenever possible)



Example: MyDelegate accelerates Conv2D and Mean operations

Performance Considerations

- Depending on the model and the operations supported by custom delegate, the final graph may have ***one or more nodes***
- If some ops are not supported by the delegate, there will be ***more than one node*** in the final graph
- It's better to avoid multiple partitions handled by the delegate, because ***data movements*** happen at the boundaries between delegate and main graph, which results in performance overhead

TensorFlow Lite Delegates – Custom Delegates

Code Example

```
// MyDelegate implements the interface of SimpleDelegateInterface.  
// This holds the Delegate capabilities.  
class MyDelegate : public SimpleDelegateInterface {  
public:  
    bool IsNodeSupportedByDelegate(const TfLiteRegistration* registration,  
                                  const TfLiteNode* node,  
                                  TfLiteContext* context) const override {  
        // Only supports Add and Sub ops.  
        if (kTfLiteBuiltinAdd != registration->builtin_code &&  
            kTfLiteBuiltinSub != registration->builtin_code)  
            return false;  
        // This delegate only supports float32 types.  
        for (int i = 0; i < node->inputs->size; ++i) {  
            auto& tensor = context->tensors[node->inputs->data[i]];  
            if (tensor.type != kTfLiteFloat32) return false;  
        }  
        return true;  
    }  
    ...  
    std::unique_ptr<SimpleDelegateKernelInterface> CreateDelegateKernelInterface()  
        override {  
        return std::make_unique<MyDelegateKernel>();  
    }  
};
```

```
        // My delegate kernel.  
        class MyDelegateKernel : public SimpleDelegateKernelInterface {  
    public:  
        ...  
    };
```



Resources – TensorFlow Lite

- [TensorFlow Lite Guide](#)
- [TensorFlow Lite Examples and Pre-Trained Models](#)
- [TensorFlow Lite Tutorials](#)
- [TensorFlow Lite Inference](#)
- [TensorFlow Lite Optimization](#)

Summary – TensorFlow Lite

TensorFlow Lite is a set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices.

TensorFlow Lite Key Features

- *Interpreter* tuned for on-device ML.
- Diverse platform support, covering [Android](#) and [iOS](#) devices, embedded Linux, and microcontrollers.
- APIs for multiple languages including Java, Swift, Objective-C, C++, and Python.
- High performance, with [hardware acceleration](#) on supported devices, device-optimized kernels, and [pre-fused activations and biases](#).
- Model optimization tools, including [quantization](#), that can reduce size and increase performance of models without sacrificing accuracy.
- Efficient model format, using a [FlatBuffer](#) that is optimized for small size and portability.
- [Pre-trained models](#) for common machine learning tasks that can be customized to your application.
- [Samples and tutorials](#) that show you how to deploy machine learning models on supported platforms.

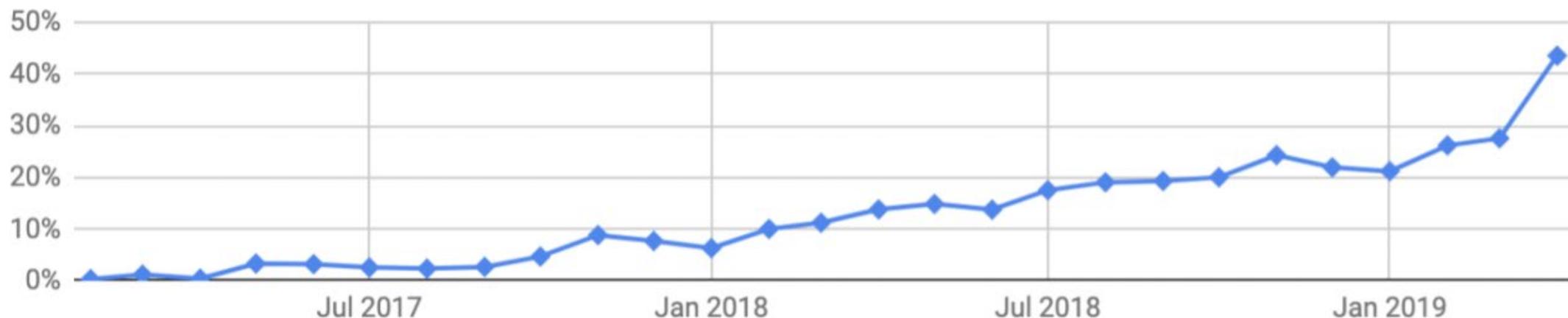
PyTorch

- Introduction
- Quick overview of building DNNs in PyTorch
- Comparison against TensorFlow



PyTorch

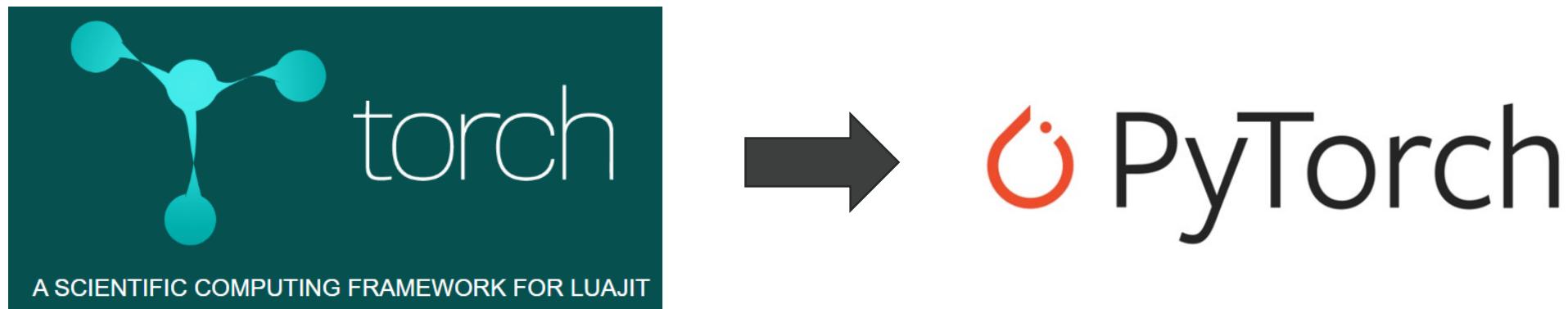
- PyTorch is a Python-based scientific computing package targeted at two sets of audiences:
 - A replacement for NumPy to use the power of GPUs
 - A deep learning research platform that provides maximum flexibility and speed



Among arXiv papers each month that mention common deep learning frameworks, percentage of them that mention PyTorch. (From the figure 3 in “PyTorch: An Imperative Style, High-Performance Deep Learning Library”)

PyTorch

- Brief history
 - Before PyTorch was created, there was and still is another framework called Torch
 - Torch is an open source library for machine learning and scientific computing based on the [Lua programming language](#)
 - Lua is a lightweight scripting language first introduced in 1993. It supports multiple programming models; it has its origins in application extensibility. Lua is compact and written in C, which makes it able to run on constrained embedded platforms.
 - As the Lua version of Torch was aging, Torch developers started a newer version written in Python. As a result, PyTorch came to be.



PyTorch Components (1/4)

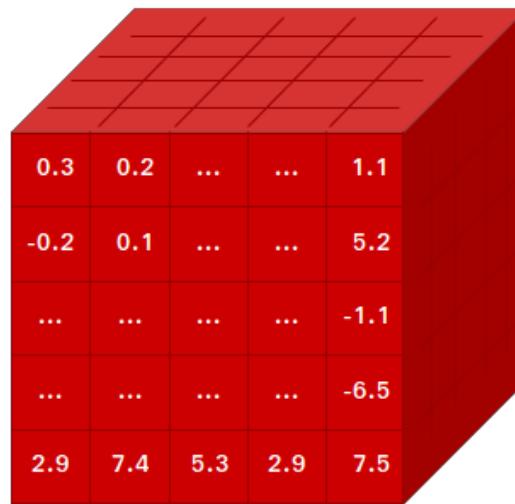
- PyTorch consists the following components:

Component	Description
<code>torch</code>	a Tensor library like NumPy, with strong GPU support
<code>torch.autograd</code>	a tape-based automatic differentiation library that supports all differentiable Tensor operations in torch
<code>torch.jit</code>	a compilation stack (TorchScript) to create serializable and optimizable models from PyTorch code
<code>torch.nn</code>	a neural networks library deeply integrated with autograd designed for maximum flexibility
<code>torch.multiprocessing</code>	Python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and Hogwild training
<code>torch.utils</code>	DataLoader and other utility functions for convenience

<https://github.com/pytorch/pytorch>

PyTorch Components (2/4)

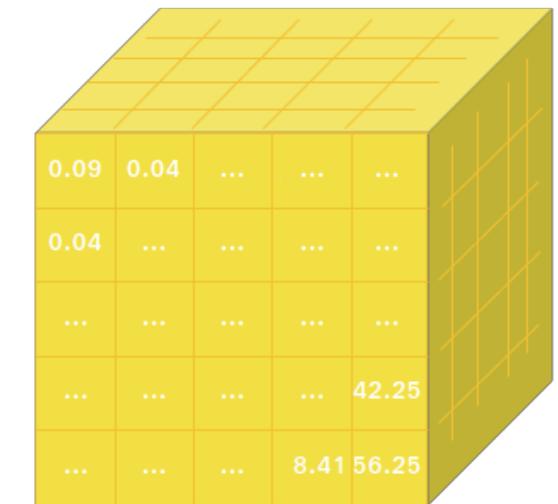
- A GPU-Ready Tensor Library
 - PyTorch provides Tensors that can live either on the CPU or the GPU, and accelerates the computation by a huge amount.



*

0.3	0.2	1.1
-0.2	0.1	5.2
...	-1.1
...	-6.5
2.9	7.4	5.3	2.9	7.5

=



<https://github.com/pytorch/pytorch>

PyTorch Components (3/4)

- Dynamic Neural Network construction

- Most frameworks such as TensorFlow / Caffe have a static view of the world. One has to build a neural network, and reuse the same structure again and again. Changing the way the network behaves means that one has to start from scratch.

In PyTorch, a technique called **reverse-mode auto-differentiation** is applied, which allows you to change neural network behaviors arbitrarily with zero lag or overhead.

- DNN graph is created on the fly
- Backpropagation uses the dynamically-created graph

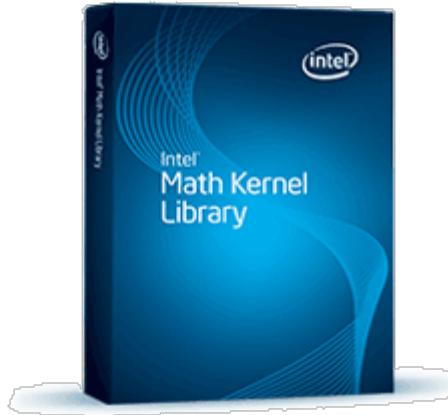
A graph is created on the fly

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```



PyTorch Components (4/4)

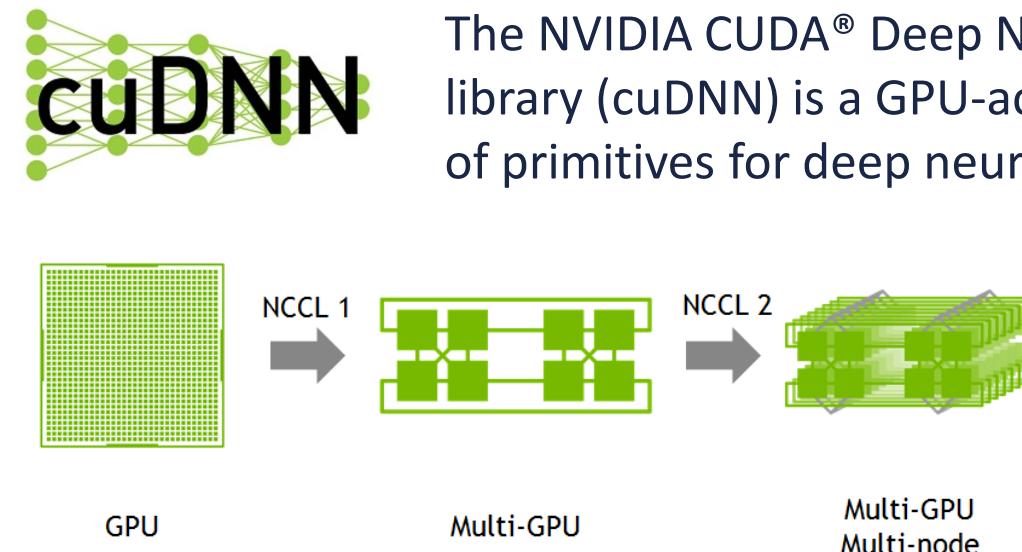
- Acceleration libraries:
 - PyTorch integrates acceleration libraries such as Intel MKL and NVIDIA cuDNN / NCCL to maximize speed. At the core, its CPU and GPU Tensor and neural network backends (TH, THC, THNN, THCUNN) are mature and have been tested for years.



Accelerate math processing routines, increase application performance, and reduce development time. This ready-to-use math library includes:

Linear Algebra | Fast Fourier Transforms (FFT) | Vector Statistics & Data Fitting | Vector Math & Miscellaneous Solvers

<https://software.intel.com/en-us/mkl>



The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks.

The NVIDIA Collective Communications Library (NCCL) implements multi-GPU and multi-node collective communication primitives that are performance optimized for NVIDIA GPUs.

<https://developer.nvidia.com>

Tensors

- Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing

Construct a randomly initialized matrix:

```
x = torch.rand(5, 3)
print(x)
```



```
tensor([[0.7799, 0.1989, 0.5753],
        [0.4624, 0.5366, 0.7436],
        [0.8719, 0.9631, 0.3204],
        [0.8461, 0.0422, 0.3457],
        [0.9910, 0.8967, 0.5962]])
```

Addition operation:

```
y = torch.rand(5, 3)
```

Syntax 1 `torch.add(x, y)`

Syntax 2 `result = torch.empty(5, 3)
torch.add(x, y, out=result)`

Syntax 3 `y.add_(x)`

NumPy Array to Torch Tensor:

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
```

Torch Tensor to NumPy Array:

```
a = torch.ones(5)
b = a.numpy()
```

Autograd: Automatic Differentiation

- The autograd package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()                                out = 3 / 4 * (x + 2)2
```

Let's backprop now. Because out contains a single scalar, out.backward() is equivalent to out.backward(torch.tensor(1.)).

```
out.backward()
print(x.grad)
```



```
tensor([[4.5000, 4.5000],
       [4.5000, 4.5000]])
```

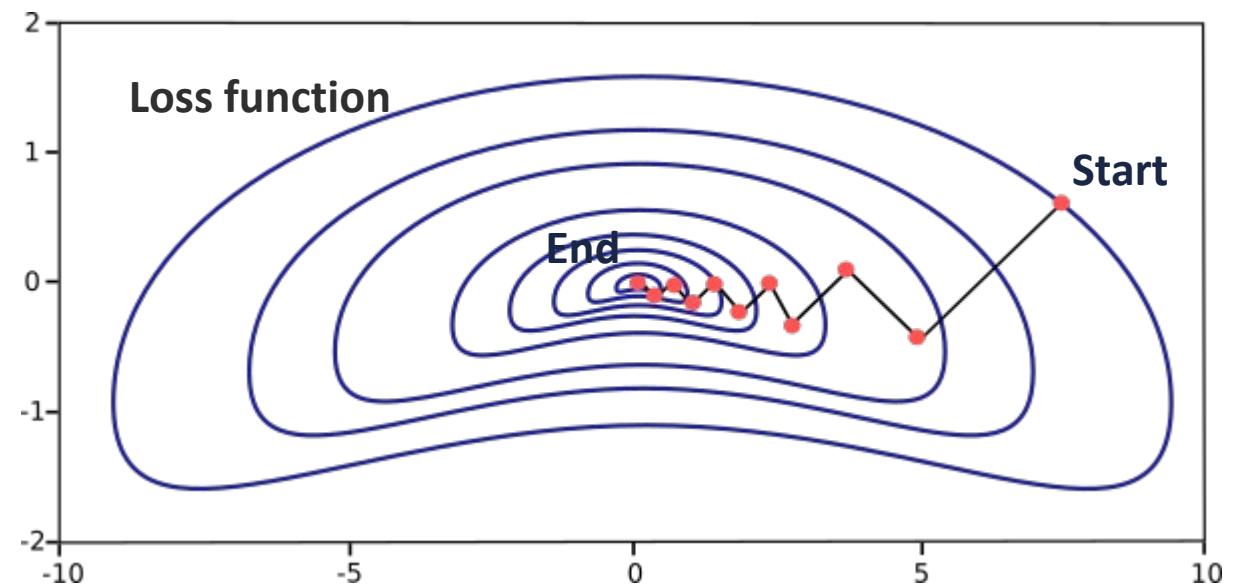
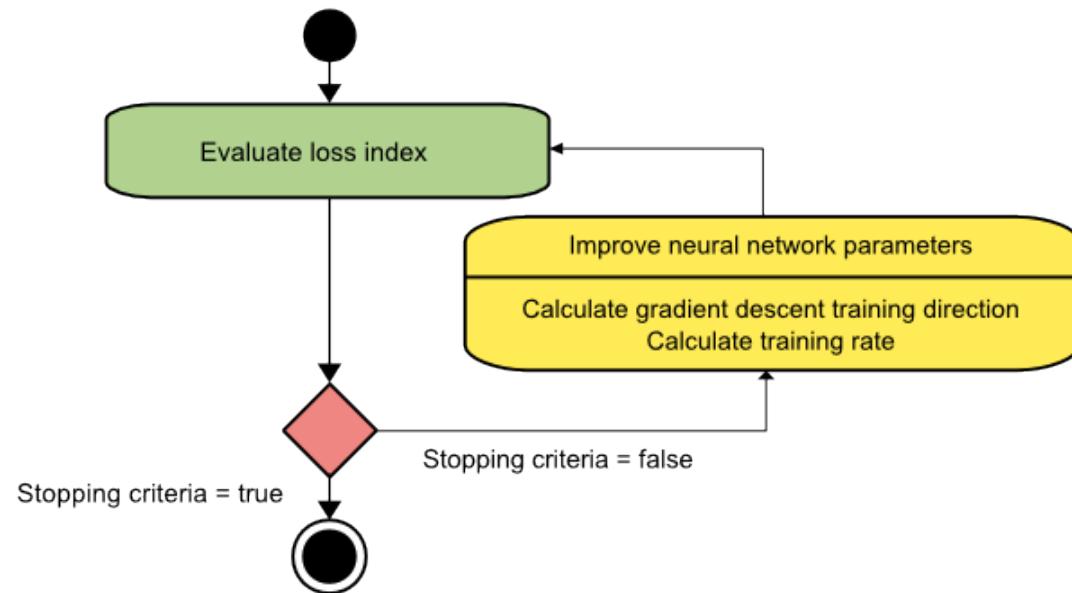
$d(out) / dx$

Gradient & Gradient Descent for DNN training

- The gradient is a vector, and each of its components is a partial derivative with respect to one specific variable.

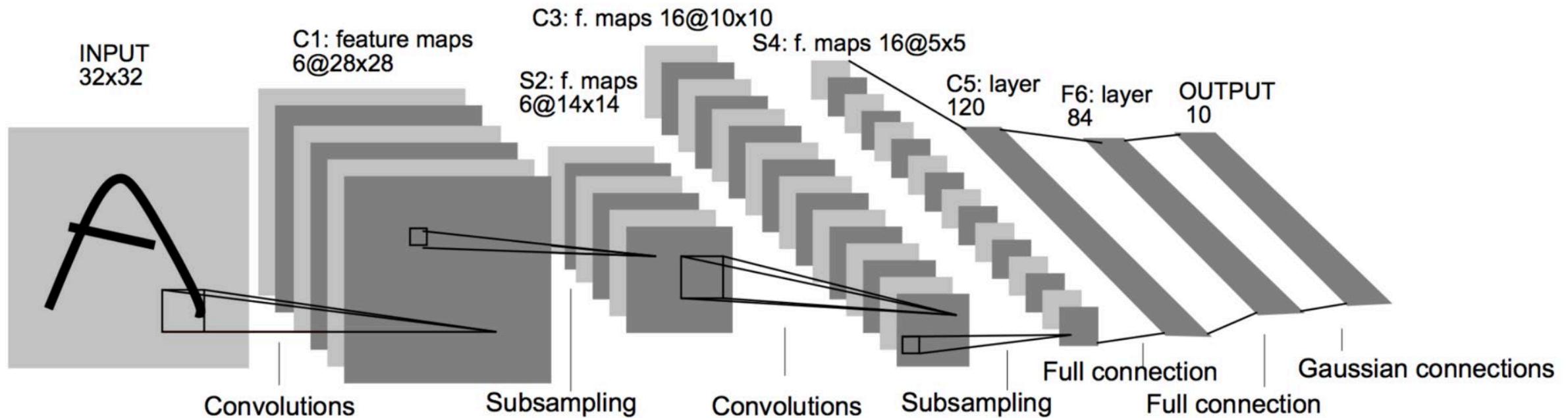
$$gradf(x_0, x_1, \dots, x_n) = \left(\frac{\partial f}{\partial x_0}, \dots, \frac{\partial f}{\partial x_j}, \dots, \frac{\partial f}{\partial x_n} \right)$$

- The gradient descent is an optimization algorithm used to minimize the loss function (evaluating the distance between the ground truth and the prediction) by iteratively updating DNN parameters in the direction of steepest descent as defined by the negative of the gradient. Less loss means better DNN performance.



Creating a neural network (1/5)

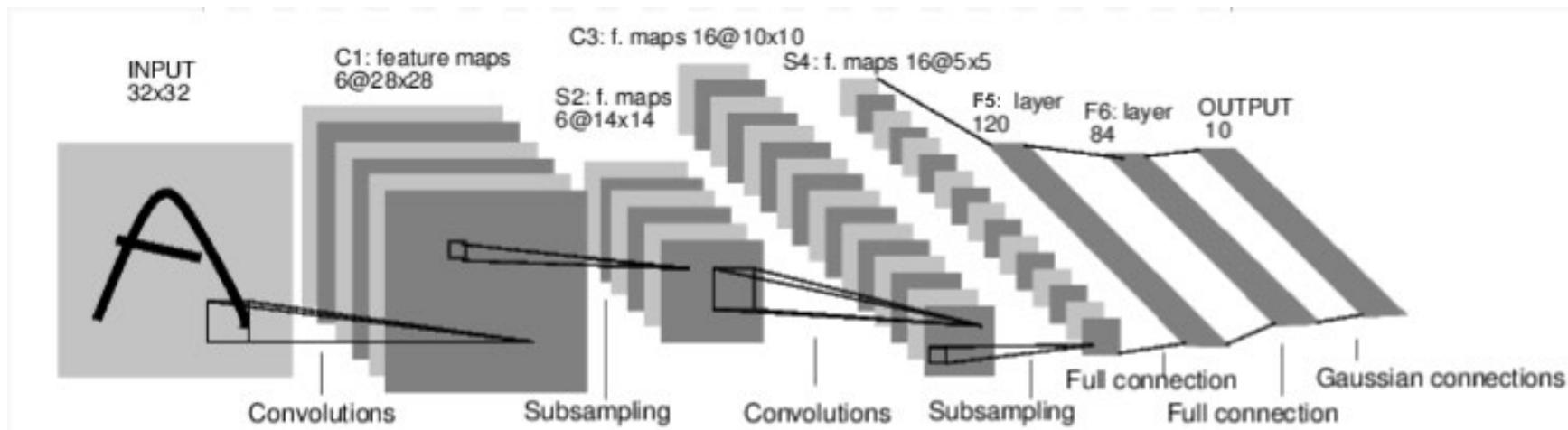
- Neural networks can be constructed using the `torch.nn` package.
 - Example: ConvNet, a network that classifies digit images



Two convolutional (CONV) layers and three fully-connected (FC) layers

Creating a neural network (2/5)

- It is a simple feed-forward network. It takes the input, feeds it through several layers one after the other, and then finally gives the output.
- A typical training procedure for a neural network is as follows:
 - Define the neural network that has some learnable parameters (or weights)
 - Iterate over a dataset of inputs
 - Process input through the network
 - Compute the loss (how far is the output from being correct)
 - Propagate gradients back into the network's parameters
 - Update the weights of the network, typically using a simple update rule: $\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$



```

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120) #
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

```

- Define the network

```

net = Net()
print(net)

```

```

Net(
    (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
    (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
    (fc1): Linear(in_features=576, out_features=120, bias=True)
    (fc2): Linear(in_features=120, out_features=84, bias=True)
    (fc3): Linear(in_features=84, out_features=10, bias=True)
)

```

- Prepare the input (random input as example) and Process input through the network

```

input = torch.randn(1, 1, 32, 32)
out = net(input)

```

Creating a neural network (4/5)

- Compute the loss

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

```
output = net(input)
target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1) # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
```



input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
-> view -> linear -> relu -> linear -> relu -> linear
-> MSELoss
-> loss

Forward

- Backprop

To backpropagate the error all we have to do is to loss.backward()

```
net.zero_grad()      # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([0.0187, -0.0201, 0.0286, -0.0095, 0.0078, -0.0092])

Creating a neural network (5/5)

- Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

PyTorch provides various different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc. To enable this, use a small package: `torch.optim` that implements all these methods.

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()    # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()          # Does the update
```

Visualizing Models/Data/Training with TensorBoard

PyTorch integrates with TensorBoard, a tool designed for visualizing the results of neural network training runs. We can intuitively inspect dataset, model architecture, training process, etc.

TensorBoard IMAGES

Show actual image size

Brightness adjustment
RESET

Contrast adjustment
RESET

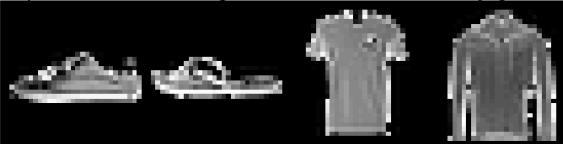
Runs
Write a regex to filter ru...
 fashion_mnist_experiment_1
TOGGLE ALL RUNS

runs

Filter tags (regular expressions supported)

four_fashion_mnist_images

four_fashion_mnist_images fashion_mnist_experiment_1
step 0 Sun Aug 04 2019 08:13:43 Pacific Daylight Time



TensorBoard SCALARS IMAGES GRAPHS PROJECTOR

Show data download links
 Ignore outliers in chart scaling

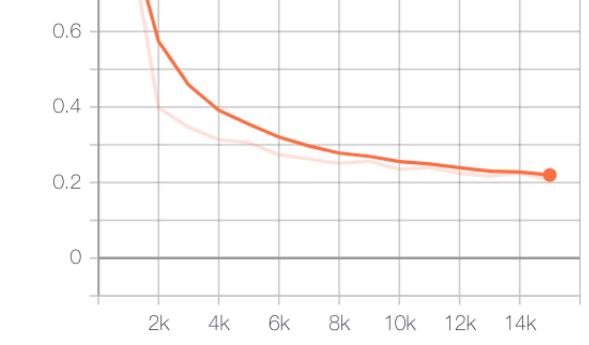
Tooltip sorting method: default

Smoothing
Horizontal Axis
STEP RELATIVE WALL

Filter tags (regular expressions supported)

training_loss

training_loss

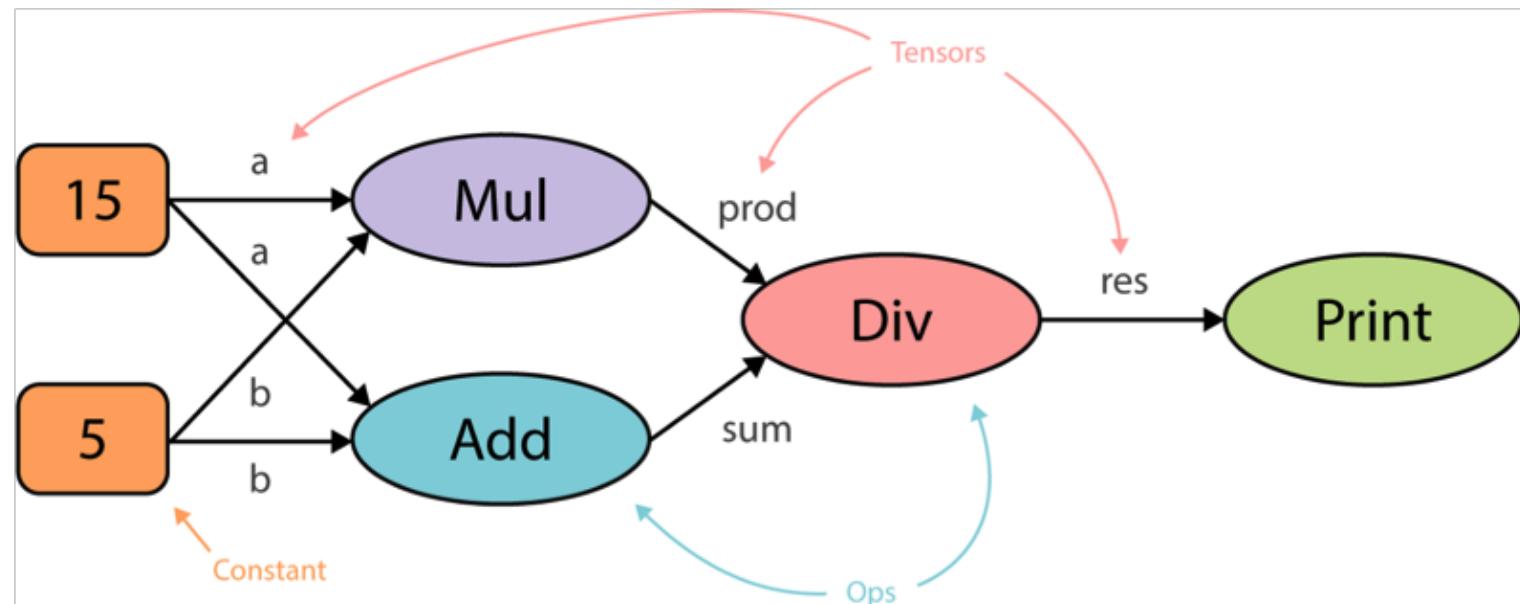


Differences: PyTorch vs TensorFlow (1/2)

1. Dynamic VS Static graph definition

- TensorFlow is a framework composed of two core building blocks:
 - A library for defining computational graphs and runtime for executing such graphs on a variety of different hardware.
 - A computational graph

```
a = 15
b = 15
prod = a * b
sum = a + b
result = prod / sum
print(result)
```



A computational graph is generated in a static way before the code is run in TensorFlow.

Differences: PyTorch vs TensorFlow (2/2)

Dynamic VS Static graph definition

- PyTorch also has two core building blocks:
 - Imperative and dynamic building of computational graphs.
 - Autograds: Performs automatic differentiation of the dynamic graphs.
- PyTorch is more tightly integrated with the Python language (more pythonic)

A graph is created on the fly

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```



The graphs change and execute nodes as you go with no special session interfaces or placeholders.



<https://builtin.com/data-science/pytorch-vs-tensorflow>

EECS 221: Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu



The Golden Age of Compilers

in an era of Hardware/Software co-design

International Conference on
Architectural Support for Programming Languages and
Operating Systems (ASPLOS 2021)

Chris Lattner
SiFive Inc

April 19, 2021

[YouTube Video Recording](#)

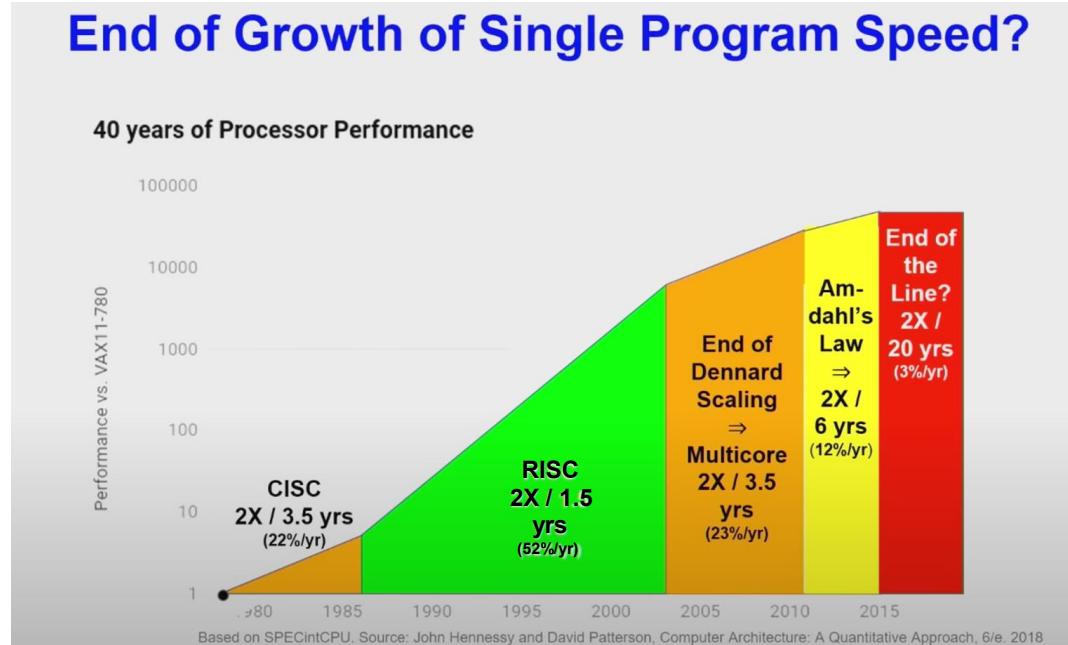
Let's talk compilers + accelerators

- Classical Compiler Design
- Modular Compiler Infrastructure
- Domain Specific Architectures
- Accelerator Compilers
- Silicon Compilers
- A Golden Age of Compilers

A New Golden Age for Computer Architecture

John L. Hennessy, David A. Patterson; June 2018

End of Growth of Single Program Speed?



What Opportunities Left?

- SW-centric
 - Modern scripting languages are interpreted, dynamically-typed and encourage reuse
 - Efficient for programmers but not for execution
- HW-centric
 - Only path left is *Domain Specific Architectures*
 - Just do a few tasks, but extremely well
- Combination
 - Domain Specific Languages & Architectures

HW / SW co-design is the best way to expose parallelism of silicon... and utilize it

Part III: DSL/DSA Summary

- Lots of opportunities
- But, new approach to computer architecture is needed.
- The Renaissance computer architecture team is vertically integrated. Understands:
 - Applications
 - DSLs and related compiler technology
 - Principles of architecture
 - Implementation technology
- Everything old is new again!



How do we program these things?

Hardware is getting harder

Modern compute acceleration platforms are multi-level and explicit:

- Scalar, SIMD/Vector, Multi-core, Multi-package, Multi-rack
- Non-coherent memory subsystems increase efficiency

Heterogeneous compute incorporating domain-specific accelerators

- Standard in high-end SoCs, domain-specific hard blocks in FPGAs

Many accelerator IPs are configurable:

- Optional extensions, tile / core count, memory hierarchy, etc



How can “normal people” write Software for this in the first place?
... and how can you *afford* to build generation-specific SW?

Next-Gen compilers and PL are needed!

We need:

- hardware abstraction spanning diverse accelerators
- support for heterogeneous compute platforms
- domain specific languages and programming models
- quality, reliability, and scalability of infrastructure

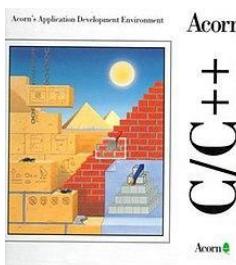
This opportunity is beckoning a *golden age* in compiler and PL technology!

Let's learn from the past, then project into the future

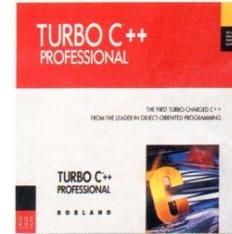
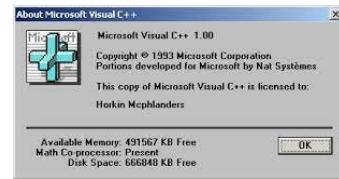
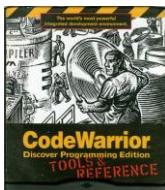
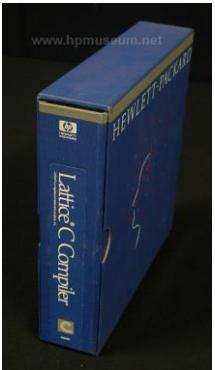


Classical Compiler Design

C Compilers leading into the early 90s



Acorn
C/C++
digital



- ⇒ Expensive, not very compatible, inconsistencies abound
- ... and didn't share any code

Also: Came in boxes, with printed manuals, often on floppy disks!

Three Phase Compiler Design

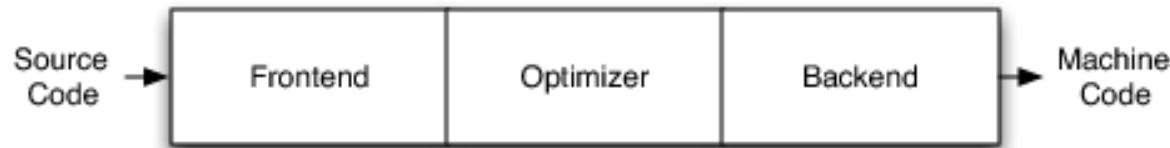


Figure 11.1: Three Major Components of a Three-Phase Compiler

FOSS Enables Collaboration & Reuse

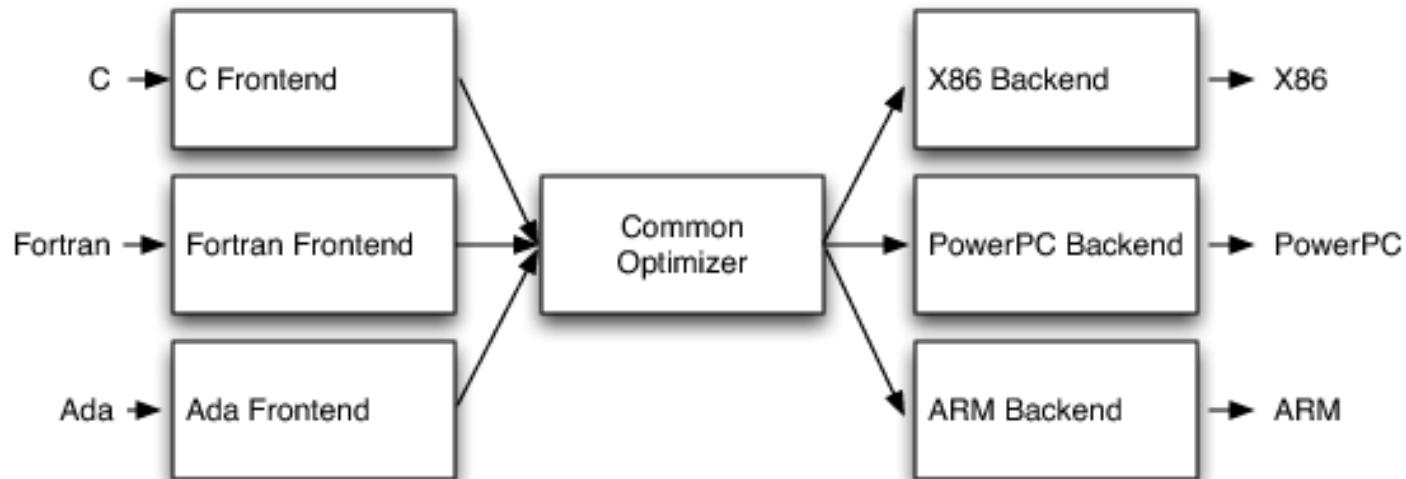


Figure 11.2: Retargetability

One frontend for many backends, one backend for many frontends

Lessons Learned

Achieved “ $O(\text{frontend} + \text{backends})$ ” scalability of compiler ecosystem

Larger center of gravity concentrated scarce compiler engineering effort

- Enables innovations in languages, frontends and backends

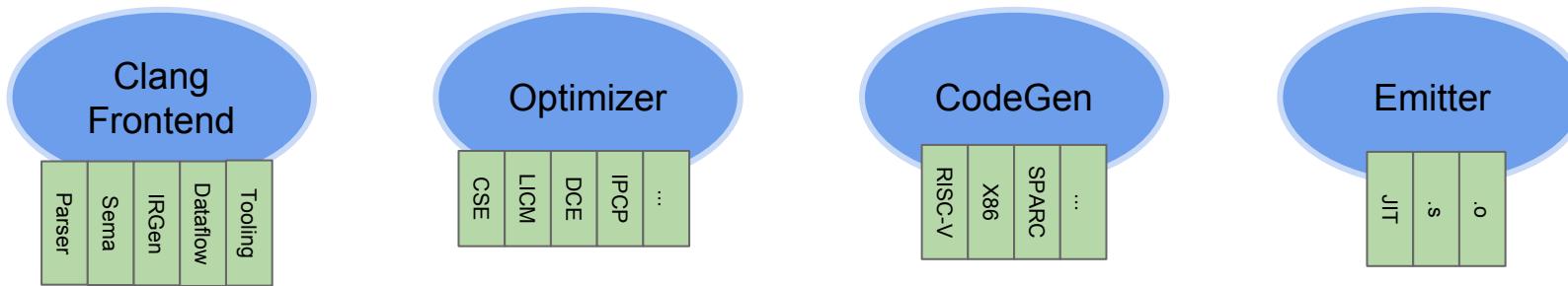
Reduced ~~fragmentation~~, standardized “C in practice”

- Enabled new business models 
- Untied the CPU ISA war from inconsequential impl details



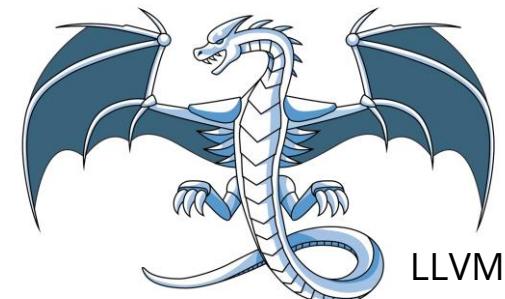
Modular Compiler Infrastructure

Library Based Design



Key insight: Compilers as libraries, not an app!

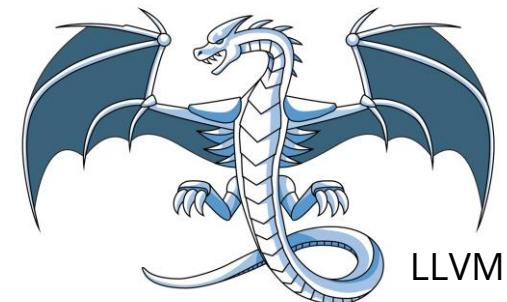
- Enable embedding in other applications
- Mix and match components
- No hard coded lowering pipeline



Components and interfaces!

Better than monolithic approaches for large scale designs:

- Easier to understand and document components
- Easier to test
- Easier to iterate and replace
- Easier to subset
- Easier to scale the community



Lessons Learned

Larger center of gravity concentrated scarce compiler engineering effort

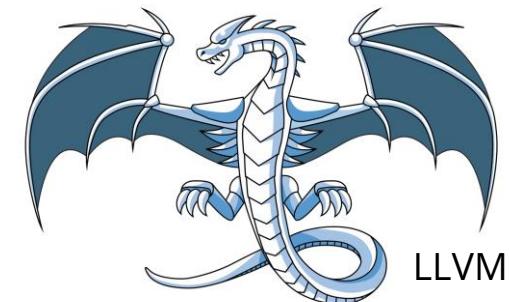
- Enables innovations in languages, frontends and backends

Reduced ~~fragmentation~~**fragmentation** of JIT compilers, standardized CPU codegen

- Enabled new business models
- Databases, graphics shader compilers, GPGPU, EDA HLS tools, ...

Scalable community architecture:

- Design methodology / developer policies
- Community policies: inclusion, licensing, extensions etc



Limitations of LLVM

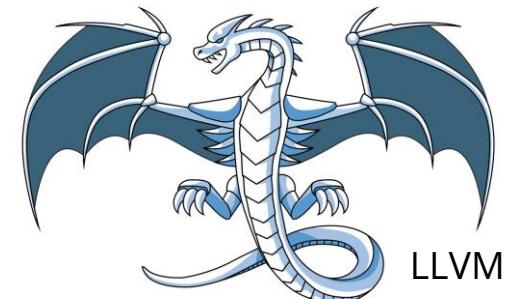
20 years in perspective on LLVM:

- “One size fits all” quickly turns into “one size fits none”
- LLVM is: 🤦 CPUs, “just ok” 💸 for SIMD, but 🤦 for many accelerators
- ... is not great for parallel programming models 🤪

Engineering is “pretty good” but could be better:

- Lots of redundancy/reimplementation @ different levels of abstraction
- Deeper discussion @ [CGO 2020 talk](#)

Going beyond basic CPUs means going beyond LLVM IR!

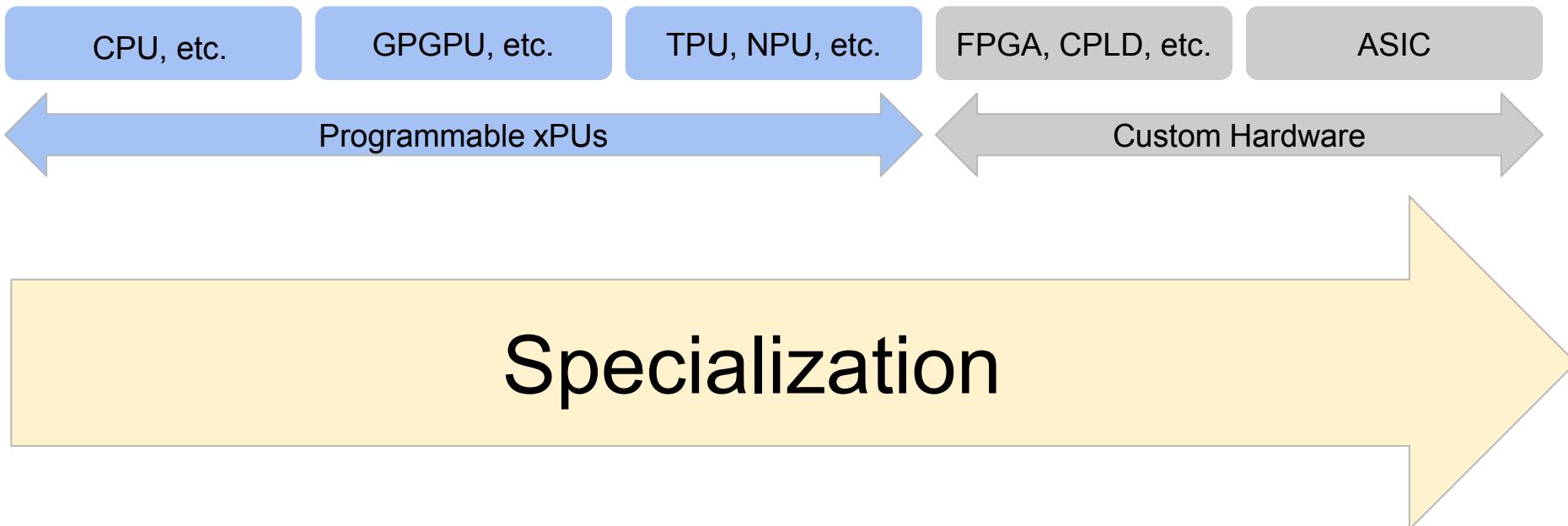


Domain Specific Architectures

Part III: DSL/DSA Summary

- Lots of opportunities
- But, new approach to computer architecture is needed.
- The Renaissance computer architecture team is vertically integrated. Understands:
 - Applications
 - DSLs and related compiler technology
 - Principles of architecture
 - Implementation technology
- Everything old is new again!

It's happening!



[[cite](#)] Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, **Mike Urbach**, MLSys Chips And Compilers Symposium 2021

Lots of players!

(an incomplete list!)

CPU, etc.



GPGPU, etc.



TPU, NPU, etc.



FPGA, CPLD, etc.



ASIC



Programmable xPUs

Custom Hardware

How do we compile for this?



cadence



AMD ROCm

SYCL™

Mentor®
A Siemens Business

tvm

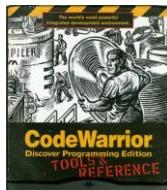
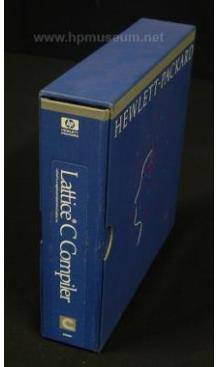
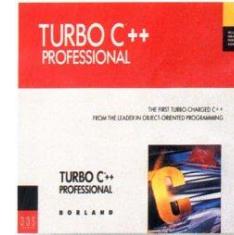
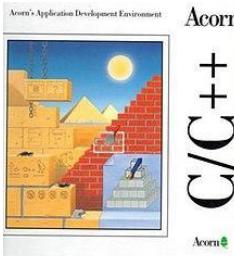
nGraph

XILINX
VITIS™
AI

VIVADO™

- ⇒ Not very compatible, inconsistent quality and scope
- ... and don't share much code

We've seen this before!



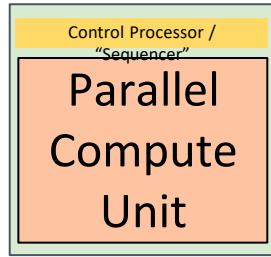
We need some unifying theories!

We need:

- “O(frontend+backends)” scalability of compiler ecosystem
- Larger center of gravity concentrated scarce compiler engineering effort
- Reduced ~~fragmentation~~**fragmentation**:
 - Ability to innovate in the programming model
 - ... without reinventing the whole stack

Accelerator Compilers

How do accelerators work?



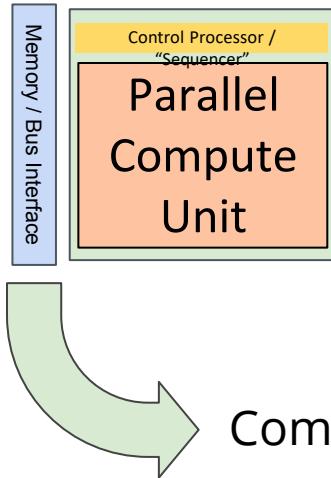
Control Processor / Sequencer

- Executes commands by the host driver app
- Handles booting and other housekeeping
- Diagnostics, security, debug, other functions

Some accelerators may do significantly more!

Ratio of control to parallel compute vary, as do the internal arch's of both

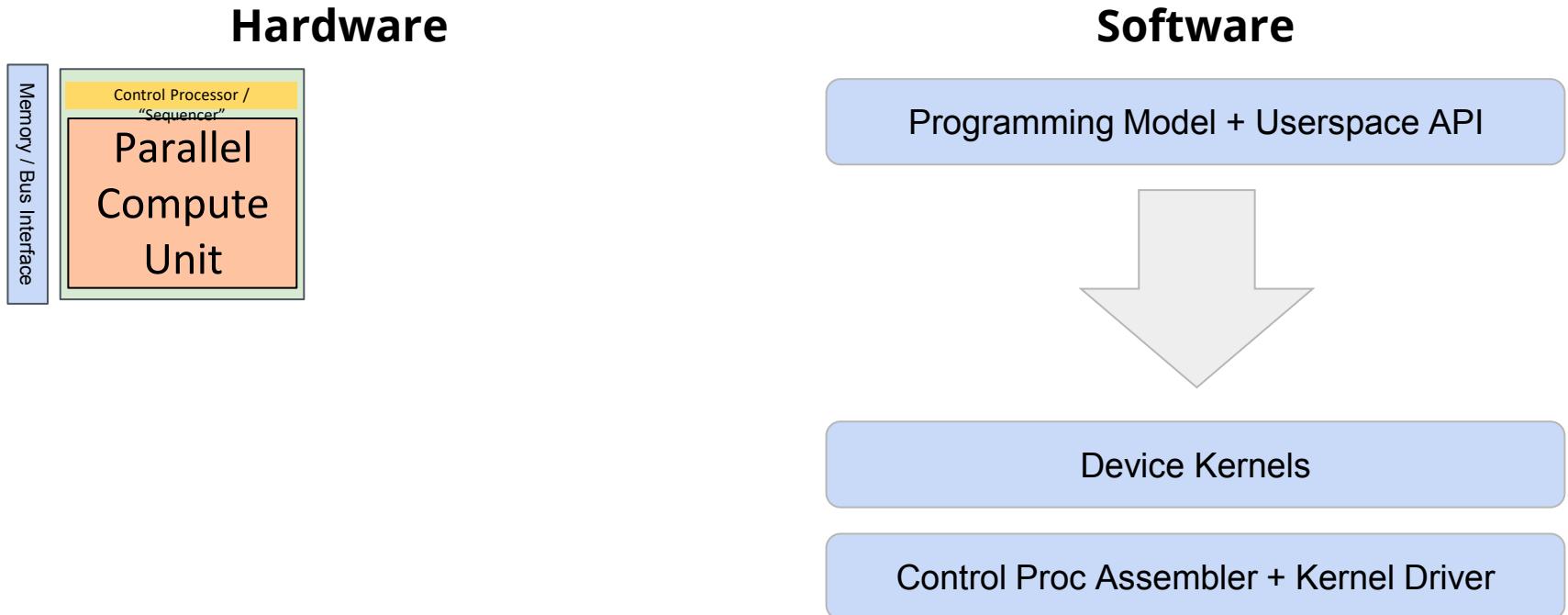
Add a system interface



Communicate w other parts of the SoC, or to off-chip resources

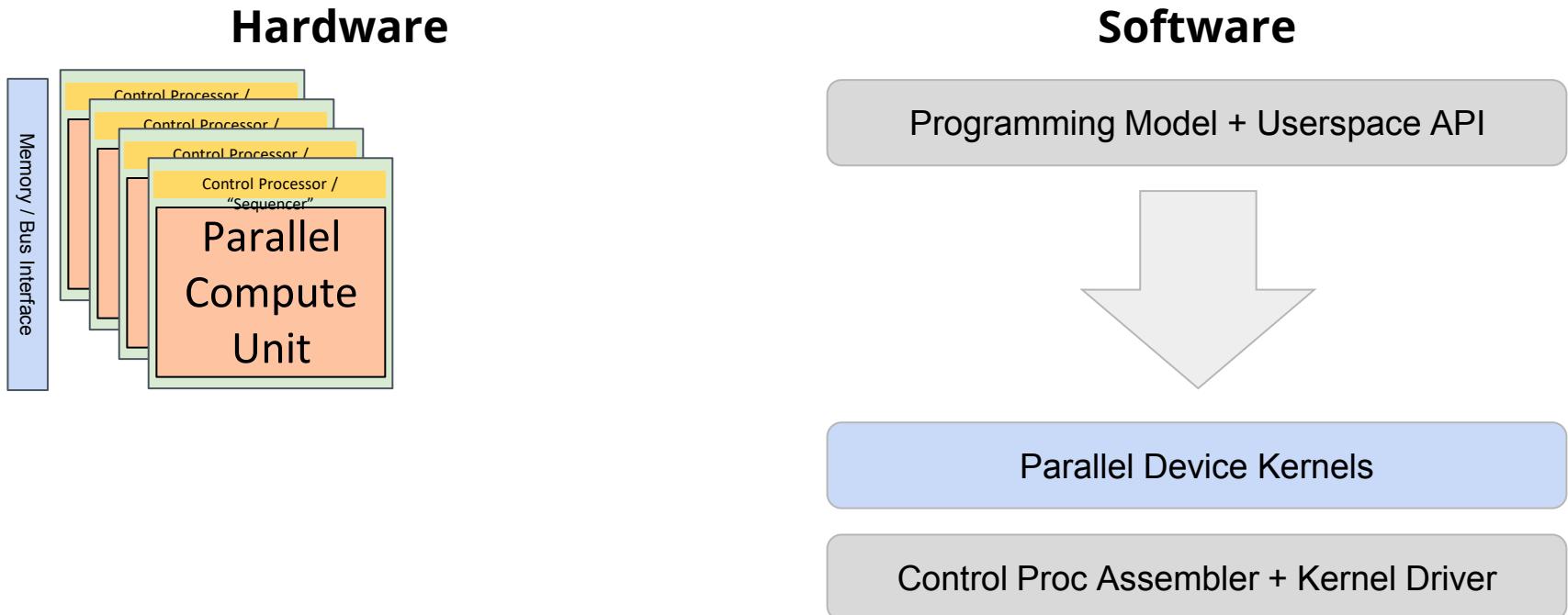
Including DDR, HBM, ... AMBA, PCI, CXL, etc depending on integration level

“Oops we need some software”



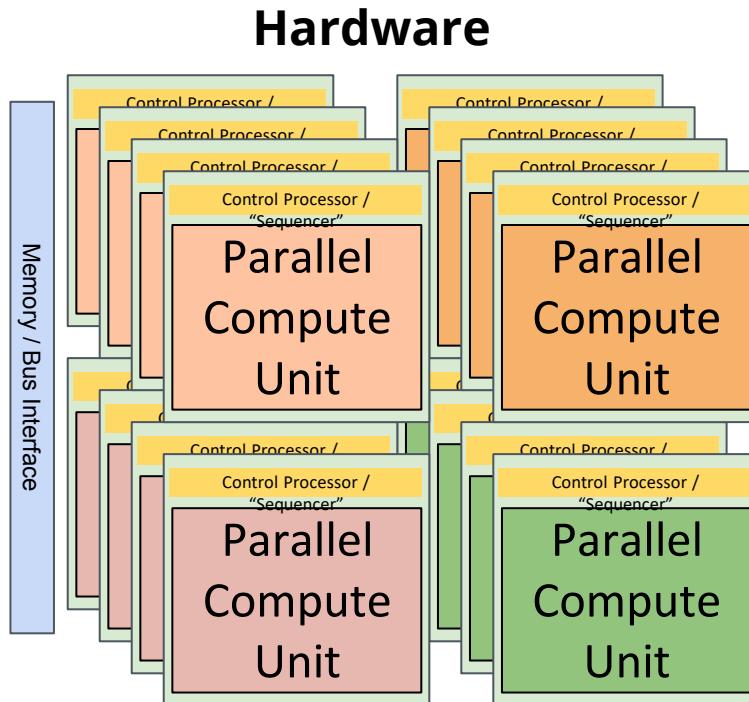
The SW people are called in after the accelerator is defined to “make it work”

Larger accelerators go multicore/SIMT...



Use of more HW area is desired, requiring parallel control logic

Tiling and heterogeneity for generality



Software

Programming Model + Userspace API

Multistream Mgmt / Interop Parallelism
Memory + Communication Optimization
Heterogenous Device + Host fallback

Parallel Device Kernels

Control Proc Assembler + Kernel Driver

⇒ Also, hierarchical compute at the board, rack, and datacenter level

Pro & Cons of hand written kernels

Benefits:

- Easy to get started, ability to get peak performance, hackability

Pro & Cons of hand written kernels

Benefits:

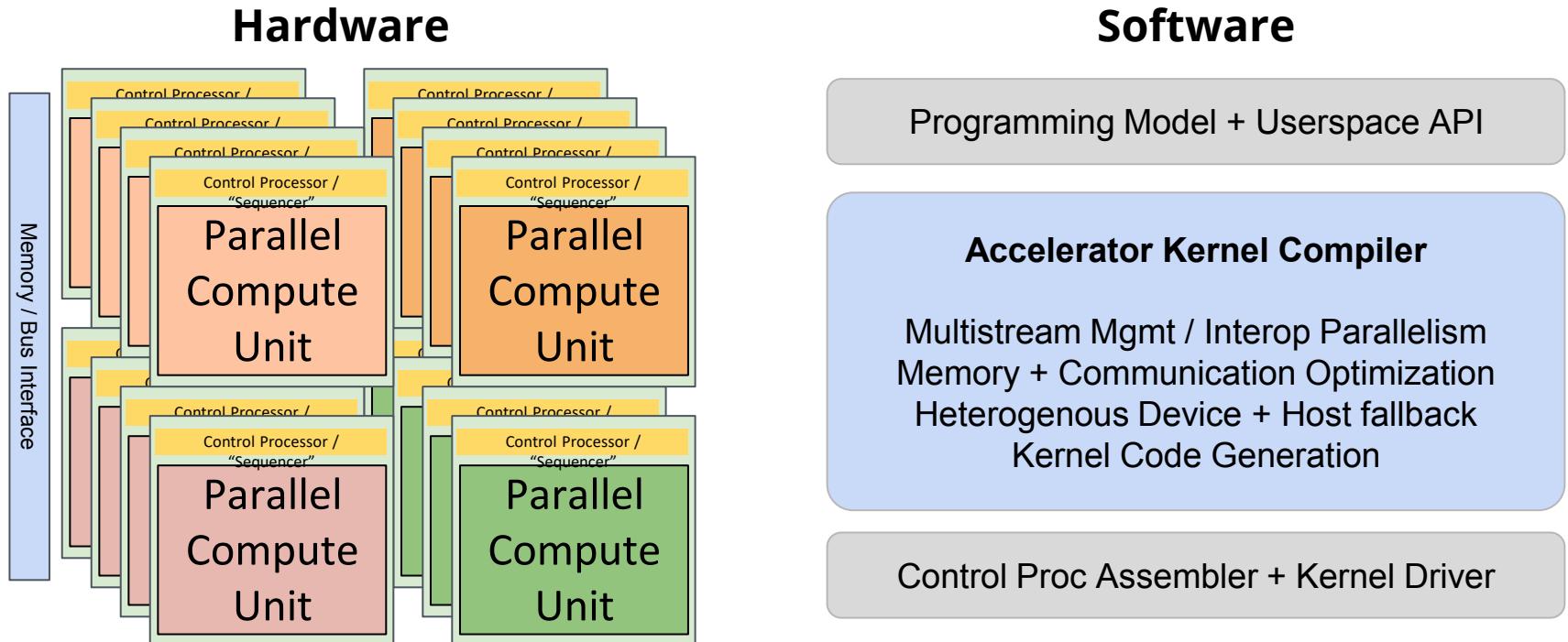
- Easy to get started, ability to get peak performance, hackability

Problem: **hand written kernels don't scale**

- Expensive to maintain a library of 100's to 1000's of kernels
- Don't scale to configurable IPs, not even memory hierarchy dimensions
- Don't scale to device families, or evolving μ arch's over time
- Eventually end up limiting HW design space exploration / evolution

Often addressed with metaprogramming (aka “mini compilers”)

“DSA Compilers” to the rescue

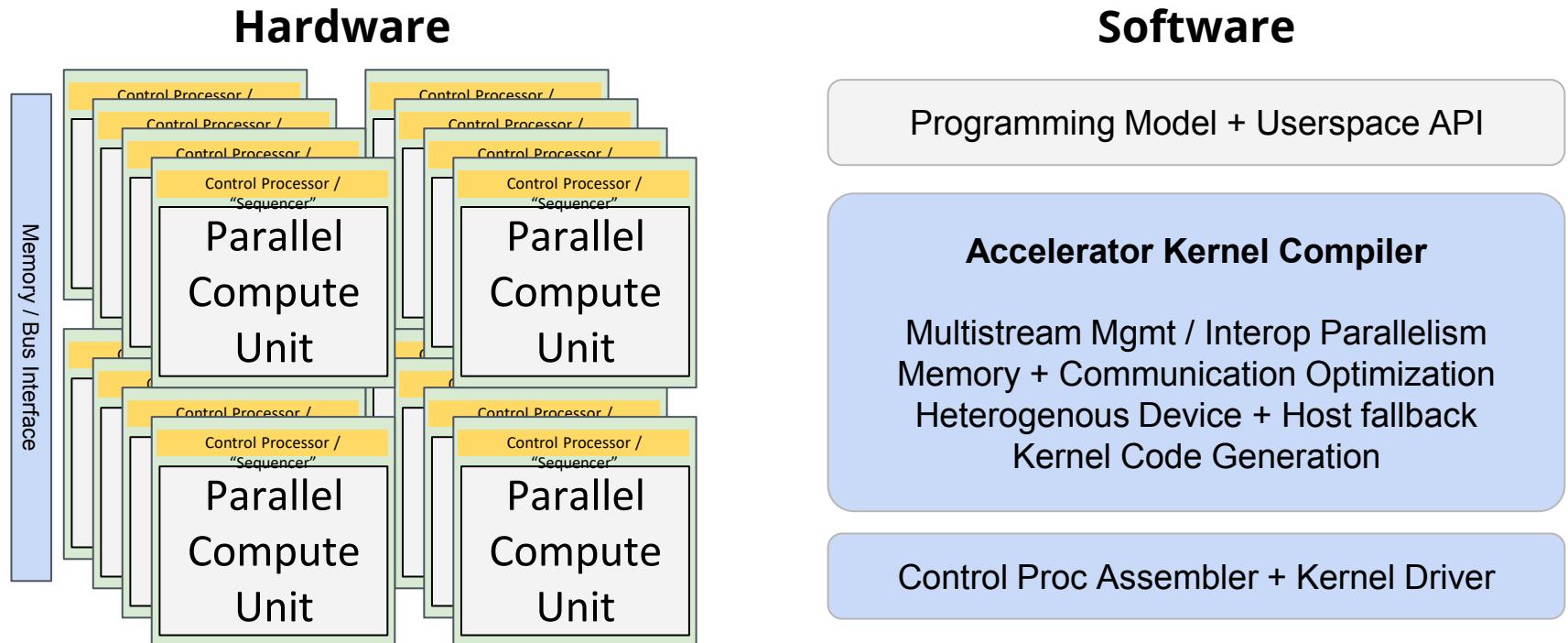


This is hard!

... and we keep reinventing it over and over again

... at the expense of usability and quality

Mostly needless **reinvention**, not **co-design**!

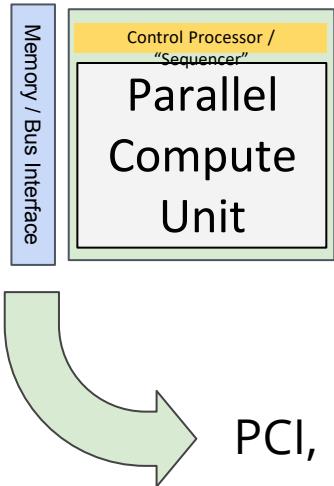


Most complexity is in non-differentiated (table stakes) components

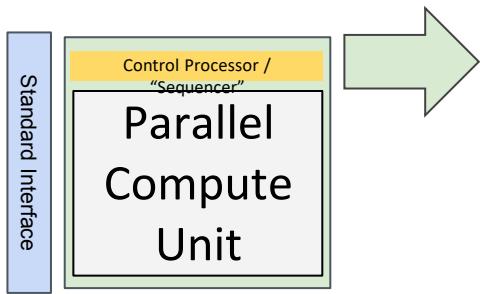
Innovate where it matters

... use open standards to accelerate the rest

Industry already standardized the buses



Standardize the Control Processor?



SW is a bigger problem than HW for accelerators:

- Control processor is bottom of the SW stack

We fool ourselves into building trivial CPUs:

- it can seem fun to design a new solution here
- ... except reset, debug, power management, security, etc (the hard parts!)

“Saving a few gates” slows down what matters

- ... and hobbles the critical path: **software**

IBM Compatibility Problem in Early 1960s

By early 1960's, *IBM had 4 incompatible lines of computers!*

701 → 7094

650 → 7074

702 → 7080

1401 → 7010

Each system had its own:

- Instruction set architecture (ISA)
- I/O system and Secondary Storage:
magnetic tapes, drums and disks
- Assemblers, compilers, libraries,...
- Market niche: business, scientific, real time, ...



IBM System/360 – one ISA to rule them all



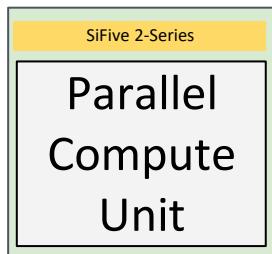
Open Industry Standard

- Many implementations available

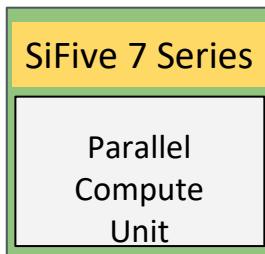
 **Modular** and subset-able ISA design:

- Extensibility allows easy addition of heterogeneous units

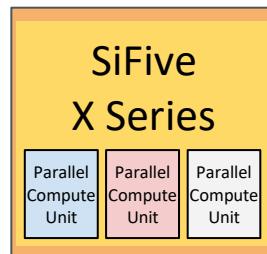
Scalability allows full spectrum of design points!



Hard Coded Accelerator



Programmable Accelerator



Heterogeneous Workload Accelerator



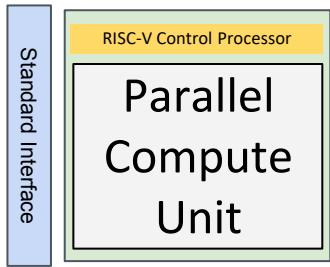
General Purpose CPU

“15,000 gates? I can’t count that low!”

Cliff Young, TPU architect, Google Brain

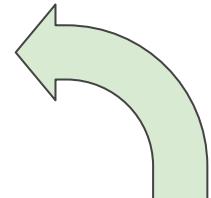
MLSys 2021 Chips and Compilers Symposium Panel

Standardize your base Software



Write your kernels in C or LLVM IR!

- Use *existing* code generators
- Use *existing* simulators
- Step through them in a **debugger**



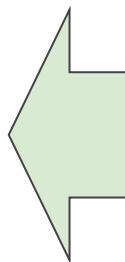
RISC-V **Compiler + Kernel Drivers**

The next frontier: DSA Compilers?

“No one size fits all” compiler!

Shape of the problem is the same...
... but the accel details always vary

How do we get reuse?



Accelerator Kernel Compiler

Multistream Mgmt / Interop Parallelism
Memory + Communication Optimization
Heterogenous Device + Host fallback
Kernel Code Generation

RISC-V Software Ecosystem

MLIR: Compiler Infra at the End of Moore's Law



- Multi-Level Intermediate Representation
- Joined LLVM, follows open library-based philosophy
-  **Modular**, extensible, general to many domains
 - Being used for CPU, GPU, TPU, FPGA, HW, quantum,
- Easy to learn, great for research
- MLIR + LLVM IR + RISC-V CodeGen = 



<https://mlir.llvm.org>

See more (e.g.):

2020 [CGO Keynote Talk Slides](#)

2021 [CGO Paper](#)

RISC-V+MLIR: Uniting an Industry

CPU, etc.



AMD

arm

intel

GPGPU, etc.



TPU, NPU, etc.



intel.

arm

XILINX

cerebras



FPGA, CPLD, etc.



AMD

intel.

ASIC



SAMSUNG

intel

Programmable xPUs

Custom Hardware

What is the benefit of this?

Larger center of gravity concentrated scarce compiler engineering effort

- Enables innovations in programming models and hardware

Achieved “O(frontend+backends)” scalability of compiler ecosystem

Reduced ~~fragmentation~~, improved ~~modularity~~ **modularity**

- Focus on the differentiated parts of the stack

But... what about hardware?

HW design is fragmented too

CPU, etc.



AMD

arm

intel

GPGPU, etc.



TPU, NPU, etc.



intel.

arm

FPGA, CPLD, etc.



AMD

intel.

ASIC



SAMSUNG

intel.

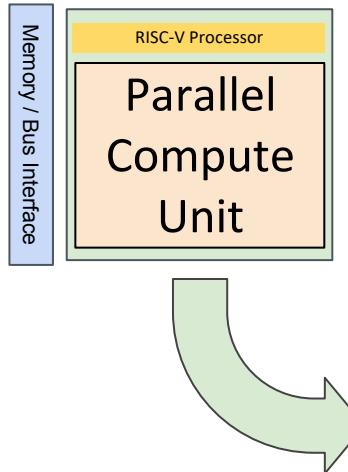


Programmable xPUs



Custom Hardware

Building Parallel Compute Units?



Notice how I conveniently omitted how to build the “interesting” part!

Silicon Compilers

Hardware Design is ripe with opportunity

SystemVerilog is industry standard, but:

- Huge, complicated, incompletely implemented
- Is it an IR? or programming language for humans? neither? both?

EDA tools are mature, but not always:

- ... innovating rapidly, now that process technology has slowed
- ... designed for usability
- ... using best practices in SW architecture
- ... cost efficient



IP-XACT



Open Source tools to the rescue?

Wonderful ecosystem of Open Source tools, but:

- Generally aspiring to be “as good” as proprietary tools
- Fragmented communities, not sharing much code
- Monolithic designs connected by unfortunate standards

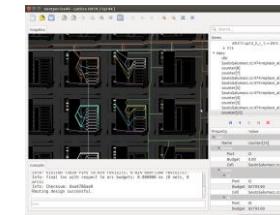
SymbiFlow

OpenROAD



Yosys

VERILATOR



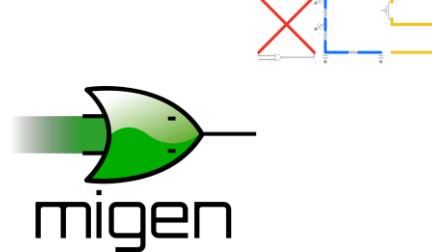
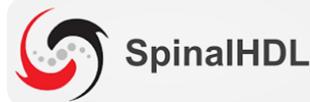
nextpnr

Innovation Explosion Underway!

Research is producing new HW design models and abstraction approaches



Dahlia



A great opportunity to pull PL + type system + compiler tech from SW world...

... held back by poor interop standards and ecosystem fragmentation

See also: [ASPLOS LATTE'21 Workshop](#)

CIRCT: Circuit IR for Compilers and Tools

Compiler infrastructure for design and verification

- LLVM incubator project built on **MLIR & LLVM**
- Composable toolchain for different aspects of hardware design / EDA processes
-  **Modularity**, library based design, ecosystem
- High quality, usability, performance

Goals:

- Unite HW design tools community
- “Accelerate” design of the accelerators!



<https://circt.llvm.org>

CIRCT Ambition / Path Ahead

Support **multiple** different “hardware design models” in one framework:

- Generators, HLS, atomic transactions, ...

Increase **abstraction level** in the hardware design IR:

- Integrate modern type system features from the SW world
- Capture more design intent, higher level verification and tools
- Better integrate formal methods into the design flow

Increase **quality** of the tools themselves:

- Compile time: shrink development cycle time
- Usability: robust location tracking for good error messages

“**10x**” design and **verification**, change economics of hardware design

Co-design of HW and SW design

CPU, etc.



AMD

arm

intel

GPGPU, etc.



TPU, NPU, etc.



intel.

arm

XILINX

cerebras



FPGA, CPLD, etc.



MicroCHIP

AMD

intel.

ASIC



SAMSUNG

intel.

Programmable xPUs

Custom Hardware



A Golden Age of Compilers

in an era of Hardware/Software co-design

Compiler/PL tech more important than ever!

The world is evolving fast at the “End of Moore’s Law”

- Changing assumptions, expanding possibilities

HW changes require new programming models and approaches:

- ... and is validating well known but sparsely adopted techniques

We need compiler and PL experts to step up!

Get involved!

<https://mlir.llvm.org/>

<https://circuit.llvm.org/>

We're hiring!



SiFive