

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*

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

*More flexible
More general*

*More specialized
More efficient*



CPU

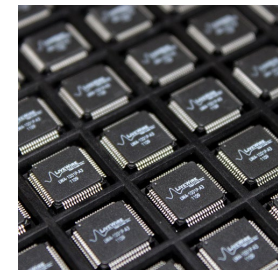


GPU



FPGA

(Field-Programmable Gate Array)

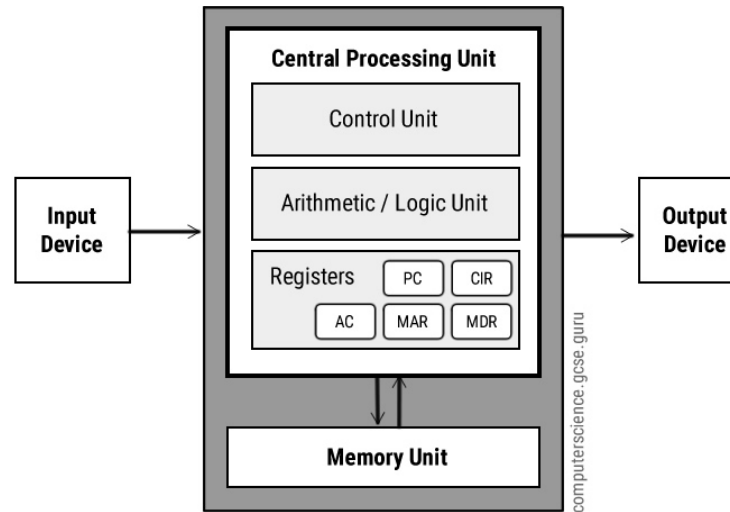


ASIC

(Application-Specific Integrated Circuit)

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
 - PC
 - CIR
 - AC
 - MAR
 - MDR
 - 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)



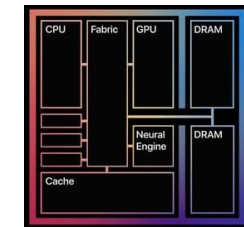
Intel i9-12900K CPU



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

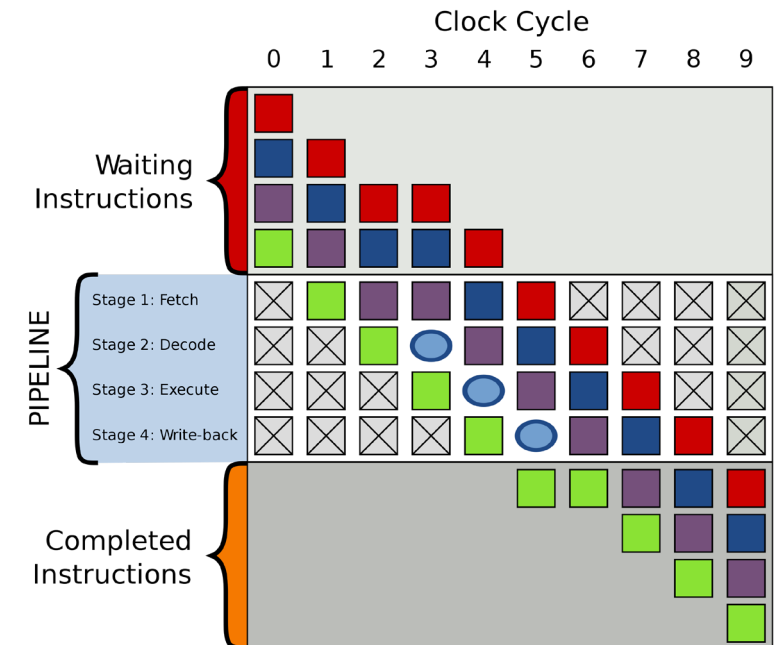
- *Data hazard*: values produced from one instruction are not available when needed by a subsequent instruction
- *Control hazard*: a branch in the control flow makes ambiguous what is the next instruction to fetch

- Pipeline Stall

- Delay in execution of an instruction in order to resolve a hazard

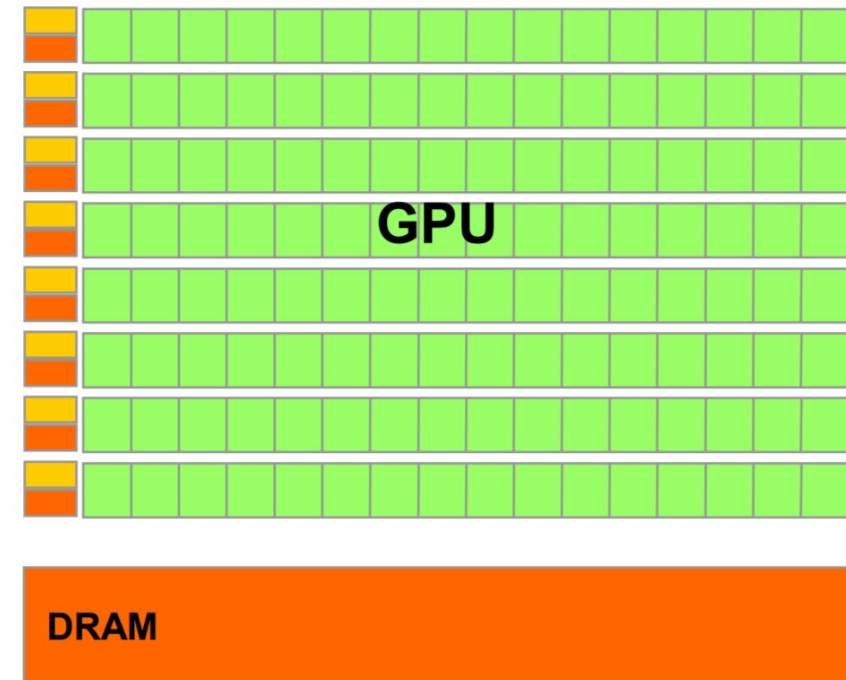
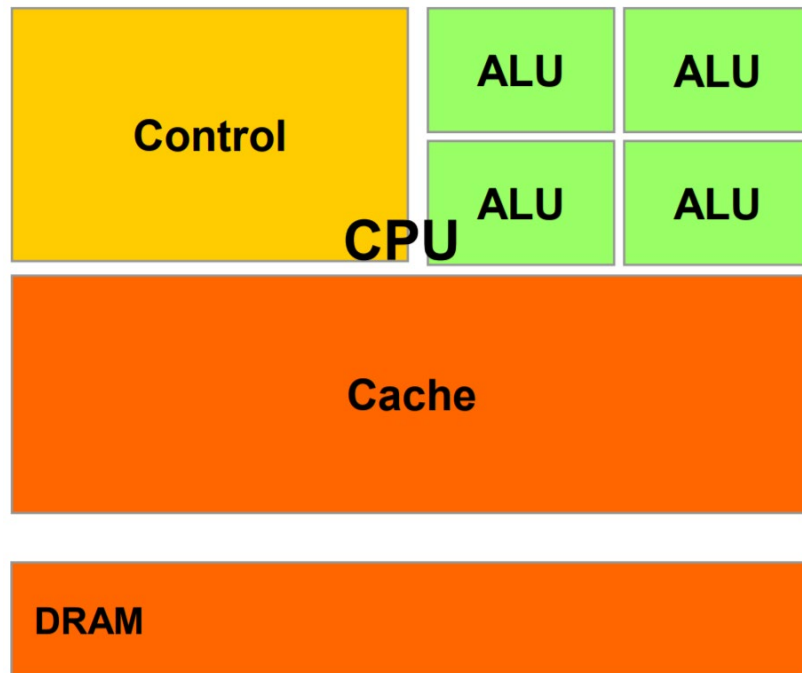
Basic five-stage pipeline

Instr. No. \ Clock cycle	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



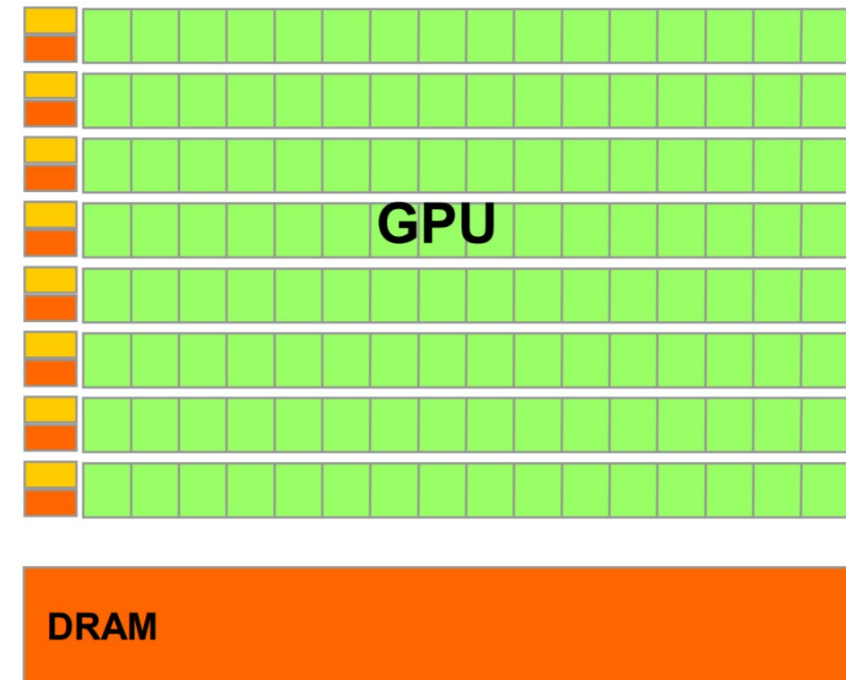
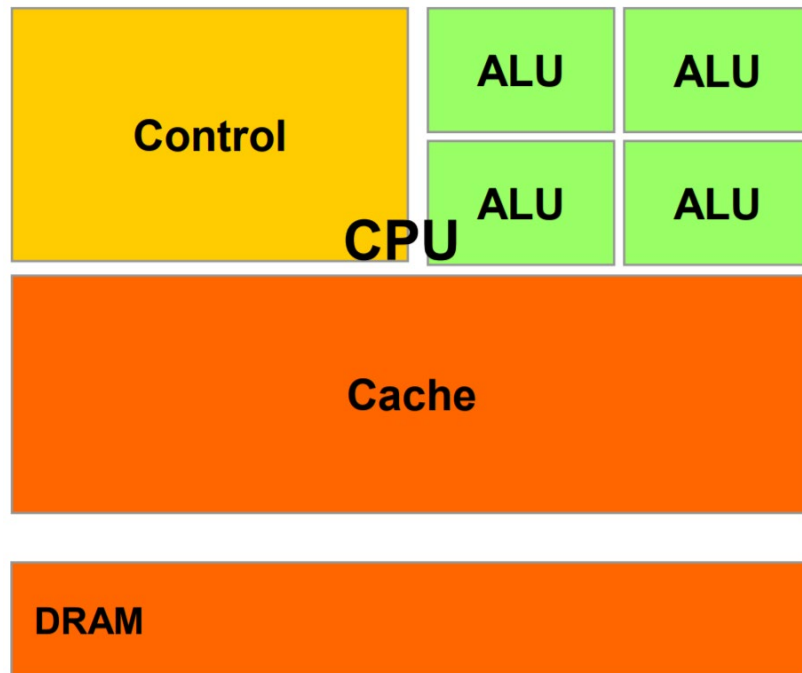
Graphics Processing Unit (GPU)

- 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**



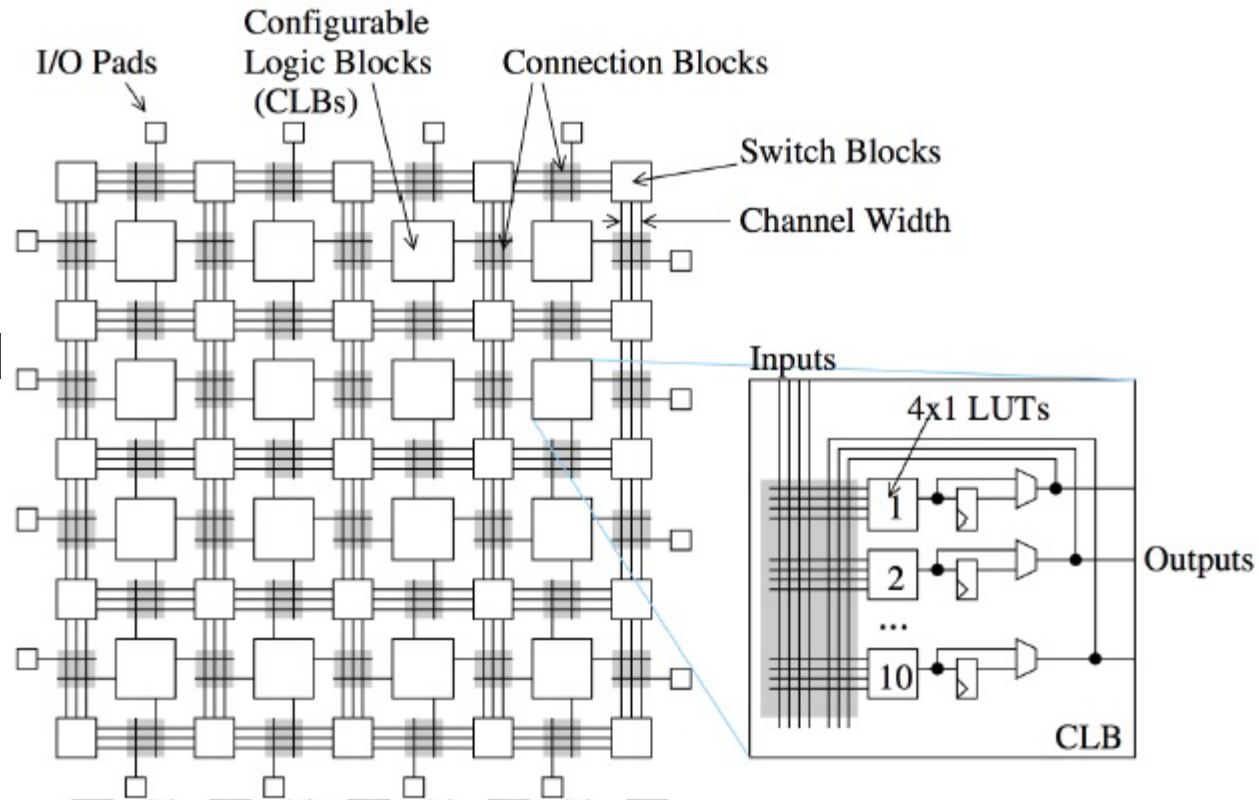
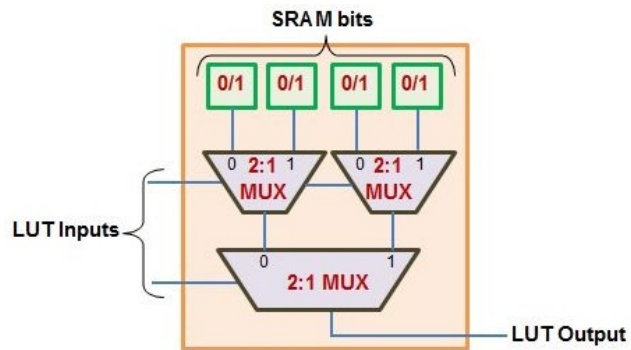
Graphics Processing Unit (GPU)

- 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)

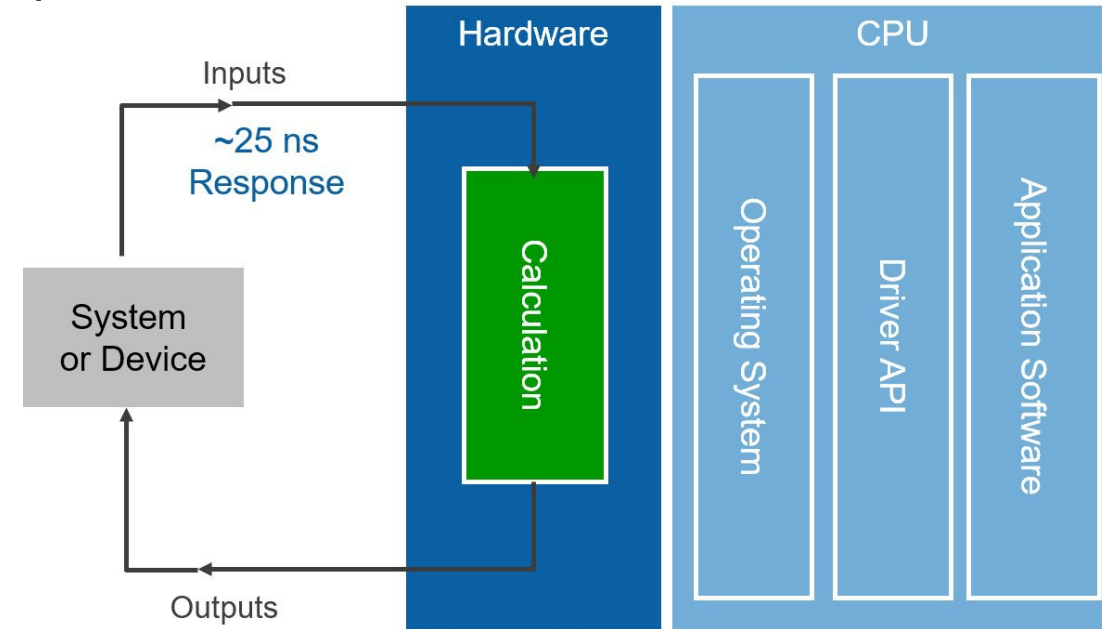
- High-density programmable gate array
- Fully programmable through bit-stream configuration file
 - Programmable Logic
 - Programmable I/O
 - Programmable Interconnects (routing)
- Look-up table (LUT) based combinational logic
 - Can be implemented with SRAM (volatile)



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)

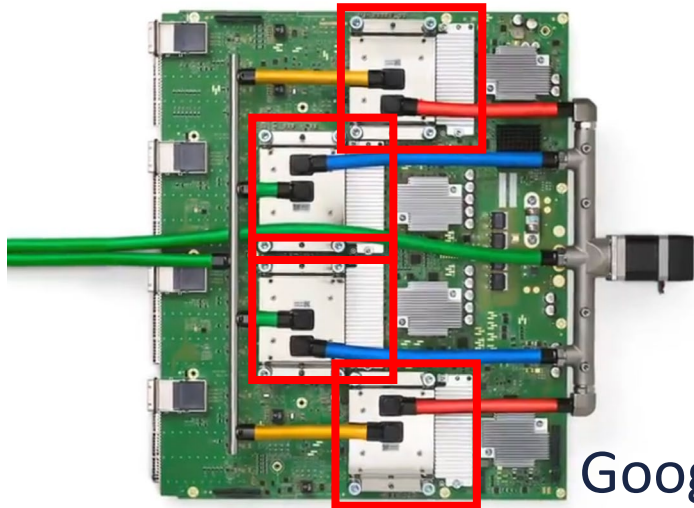
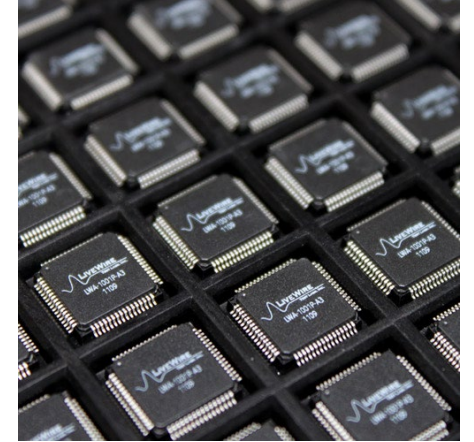
- Massive fine-grained parallelism
- Clock cycle accurate control & compute
- Very low latency, very short response time (benefits from specialization and customization)
 - Applications: real-time video processing, signal processing, high-frequency trading, etc.
- Lower clock frequency (typically < 1GHz)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



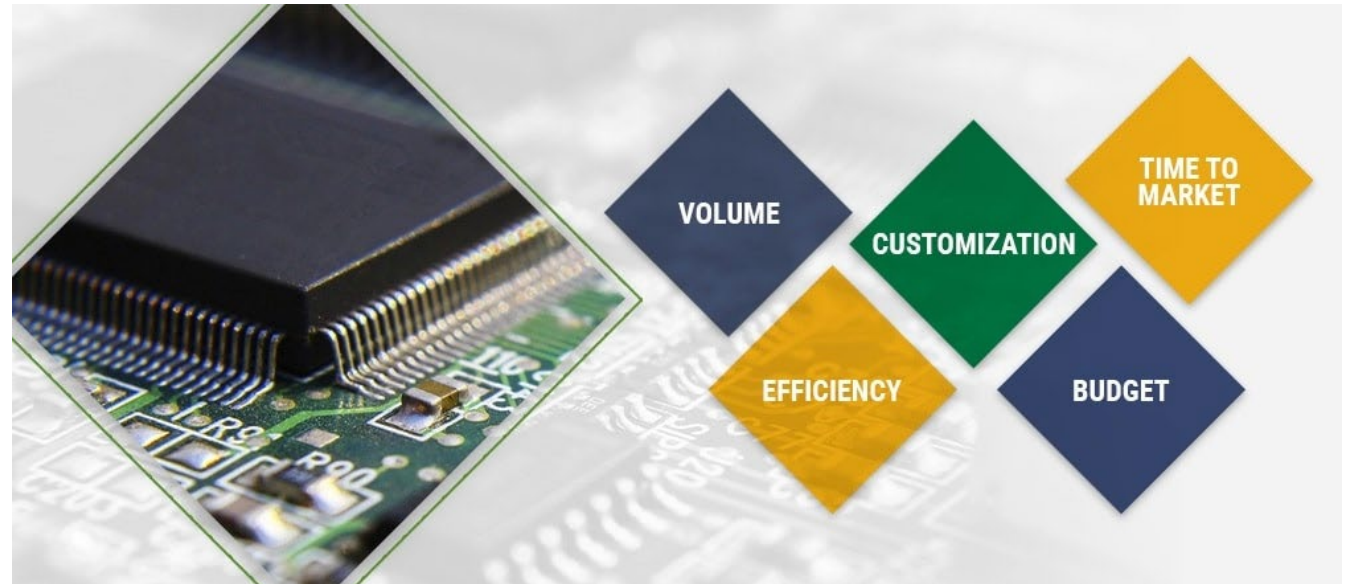
Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

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



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

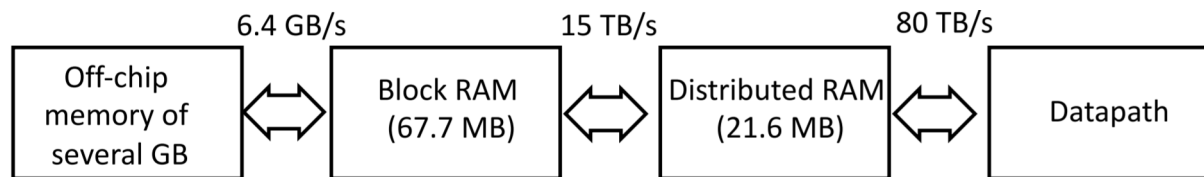
Basic Concepts

- 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.

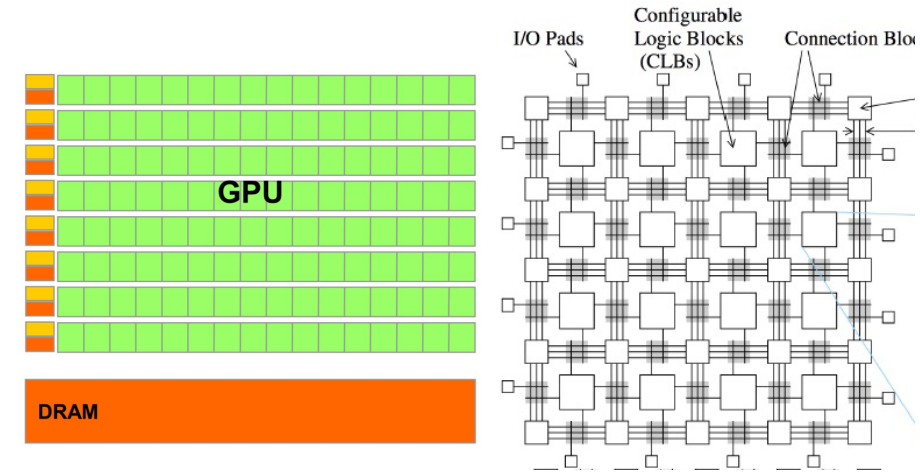


Bandwidth/memory distribution in Xilinx Virtex-7 FPGA

(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*

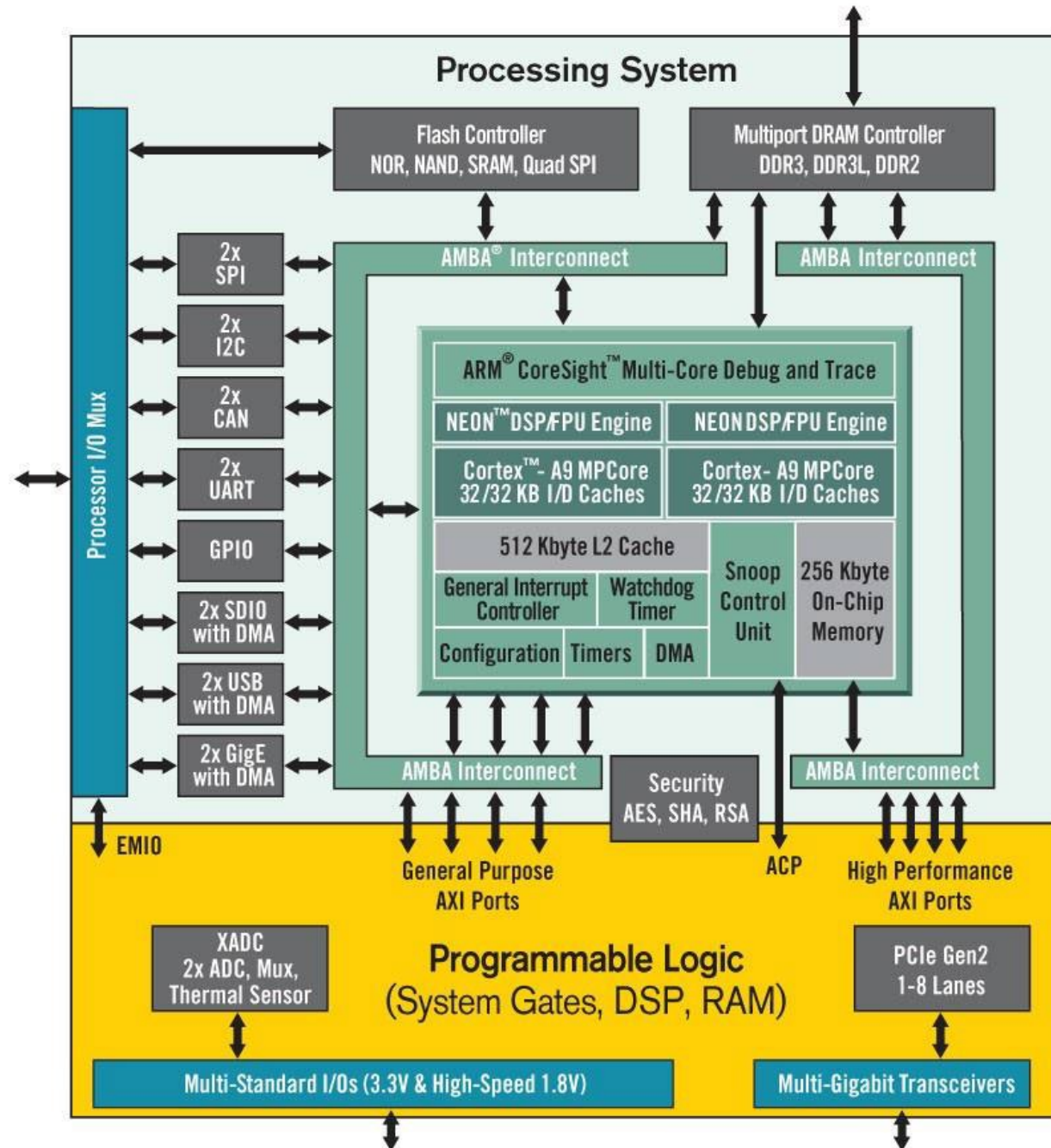
- “Compute” to “Memory” ratio:

$$\frac{\text{Number of operations (FLOPs)}}{\text{Data transferred through memory for the operations (Bytes)}}$$

- 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)

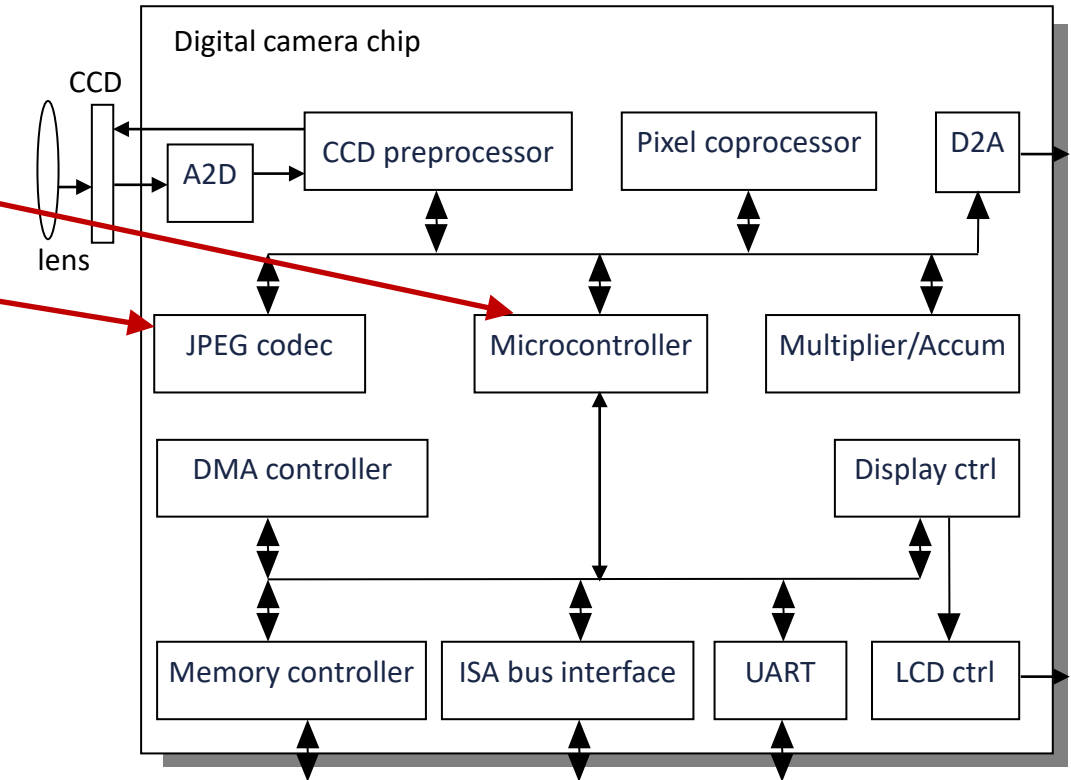
Interface Choices

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

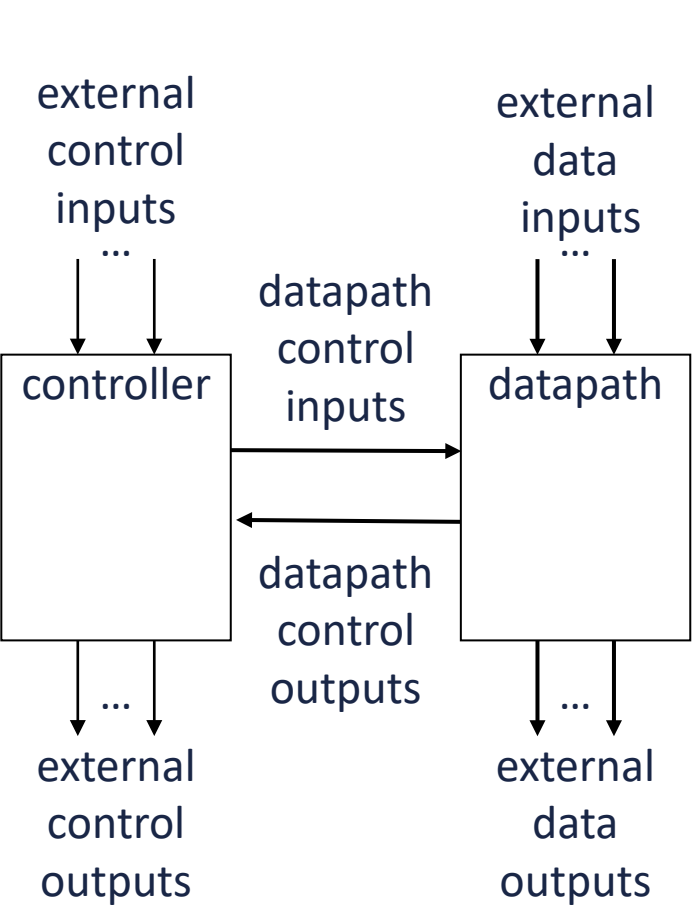


Designing Single-Purpose Processors

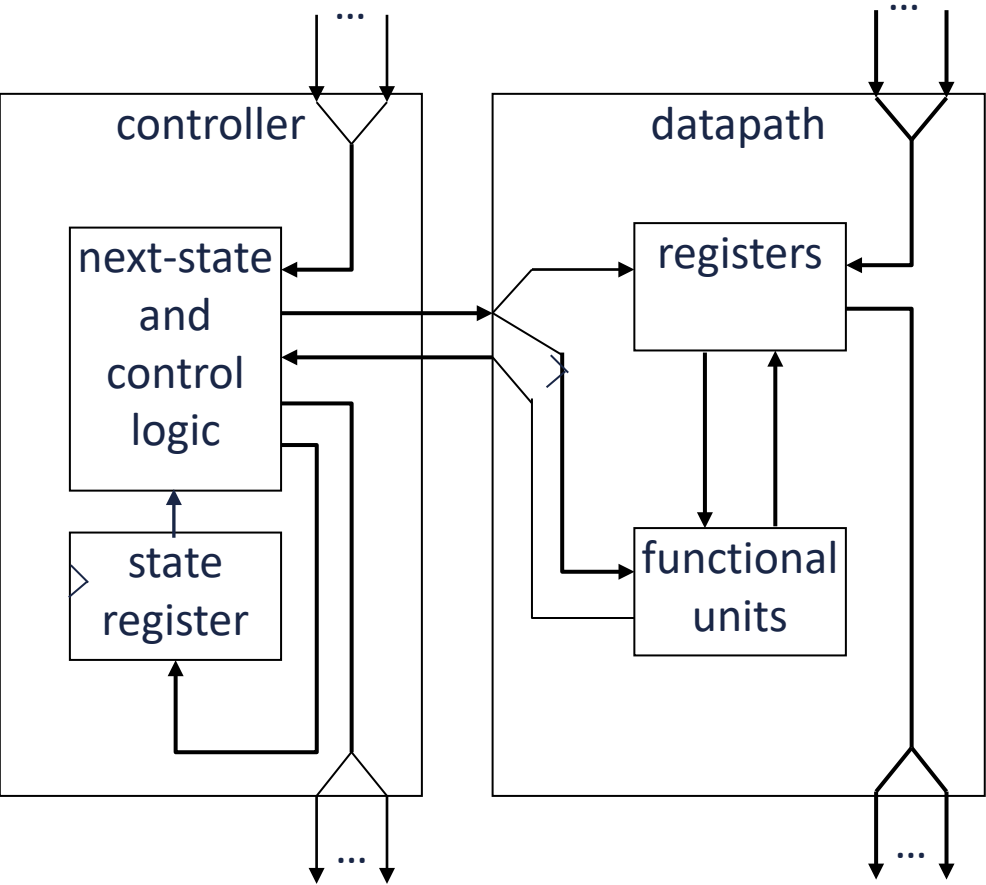
- Processor
 - Digital circuit that performs computation tasks
 - Contains controller and datapath
 - General-purpose: variety of computation tasks
 - Single-purpose: one particular computation task
 - Application-specific instruction-set processor (ASIP): domain specific tasks
- A custom single-purpose 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



controller and datapath

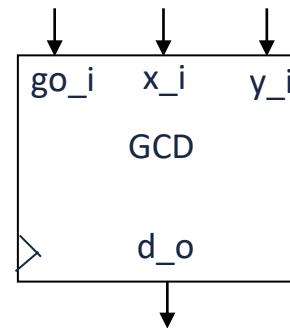


a view inside the controller and datapath

Example: Greatest Common Divisor (GCD)

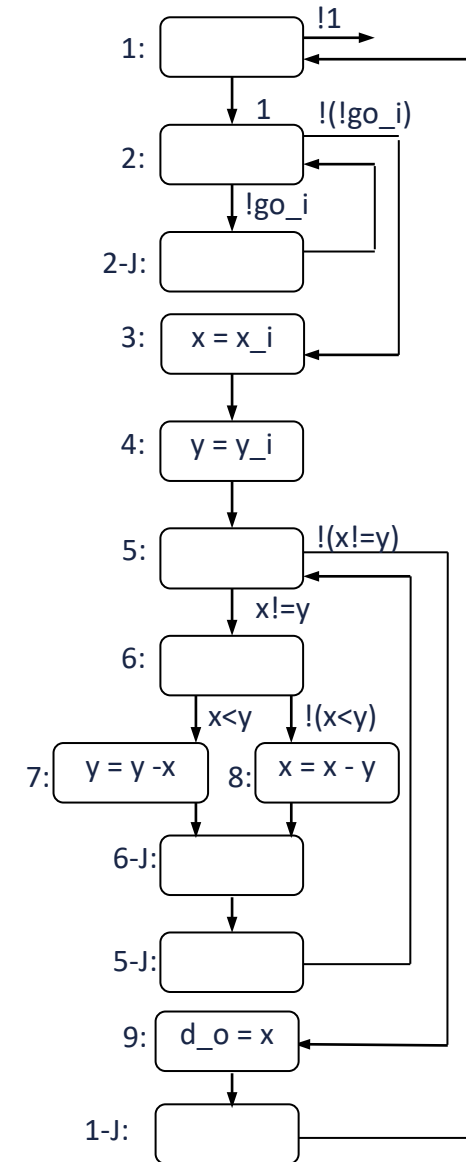
- First, create algorithm
- Then, convert algorithm to “complex” state machine
 - Known as FSMD: Finite-State Machine with Datapath
 - Can use templates to perform such conversion

(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;
8:     else
7:       x = x - y;
9:   }
9:   d_o = x;
}
```

(b) desired functionality

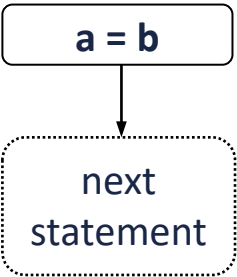


(c) FSMD

State Diagram Templates

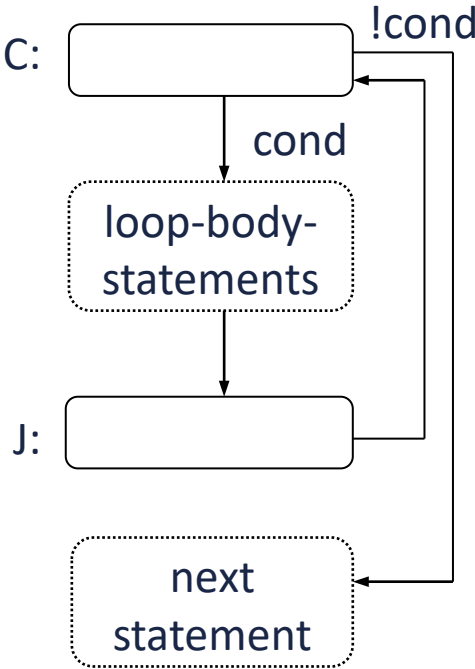
Assignment statement

a = b
next statement



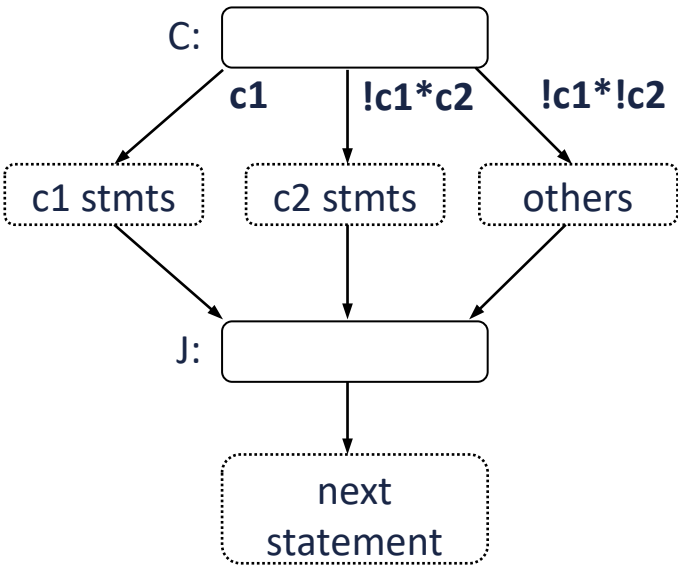
Loop statement

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



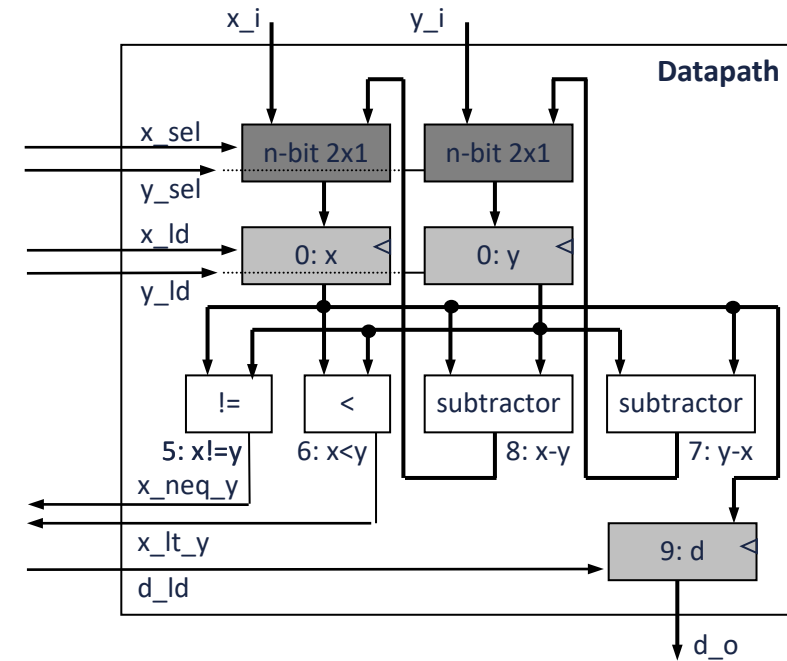
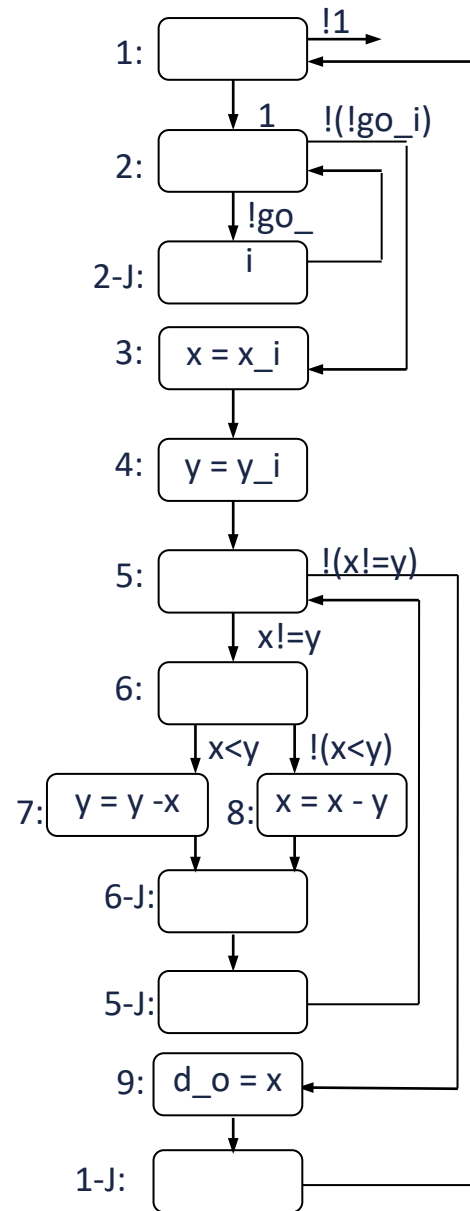
Branch statement

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

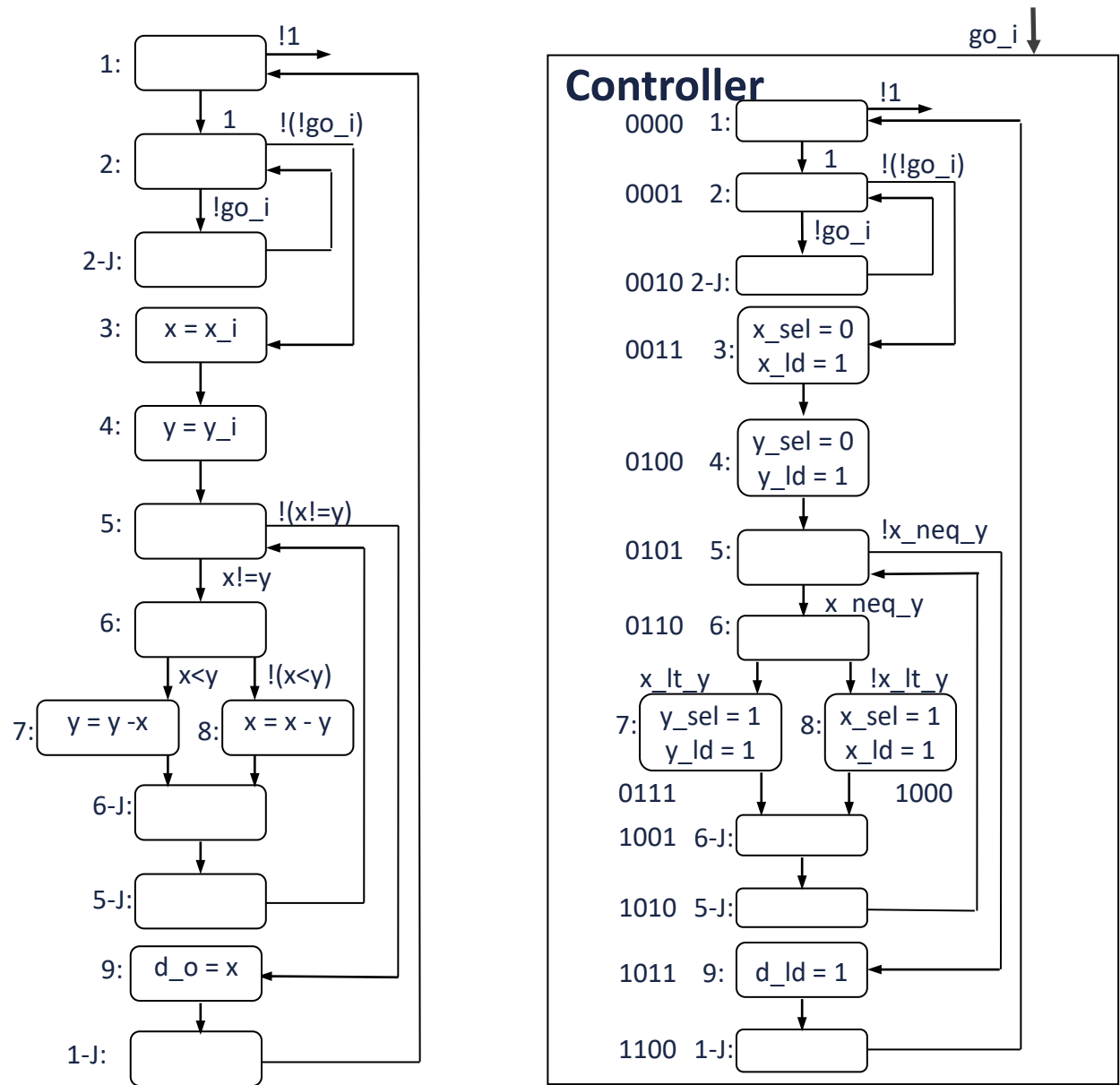


Creating the Datapath

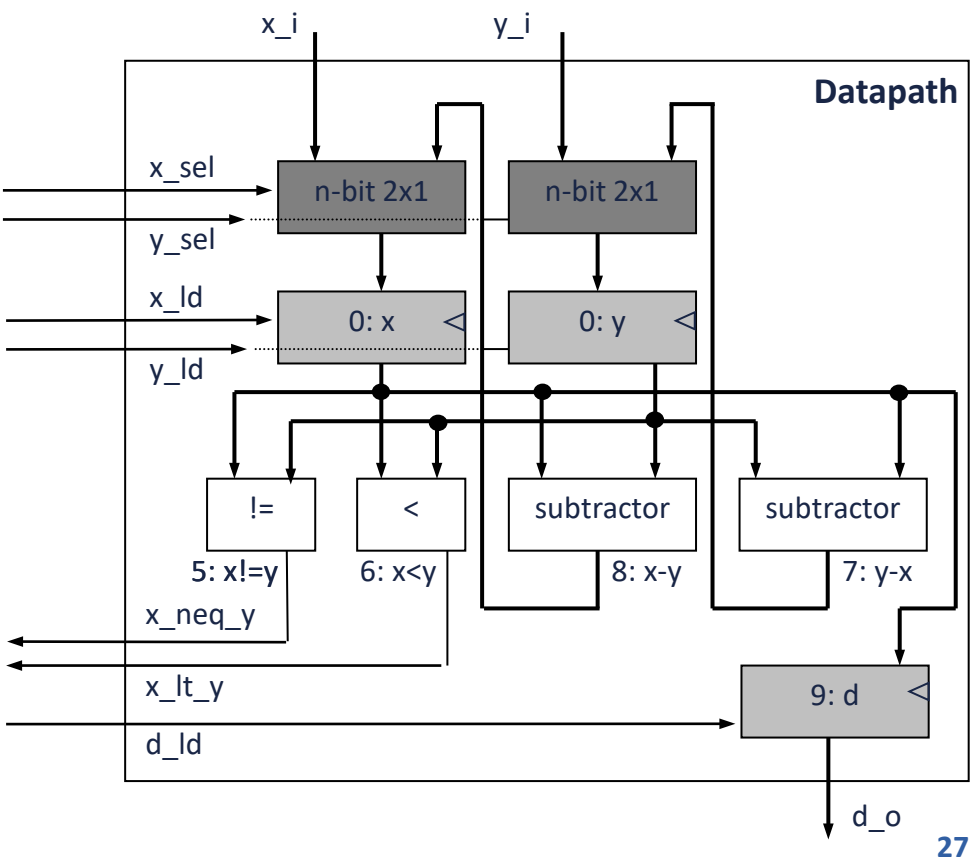
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers, and functional units
 - Based on reads and writes
 - Use multiplexors for multiple sources
- Create unique identifier
 - For each datapath component control input and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



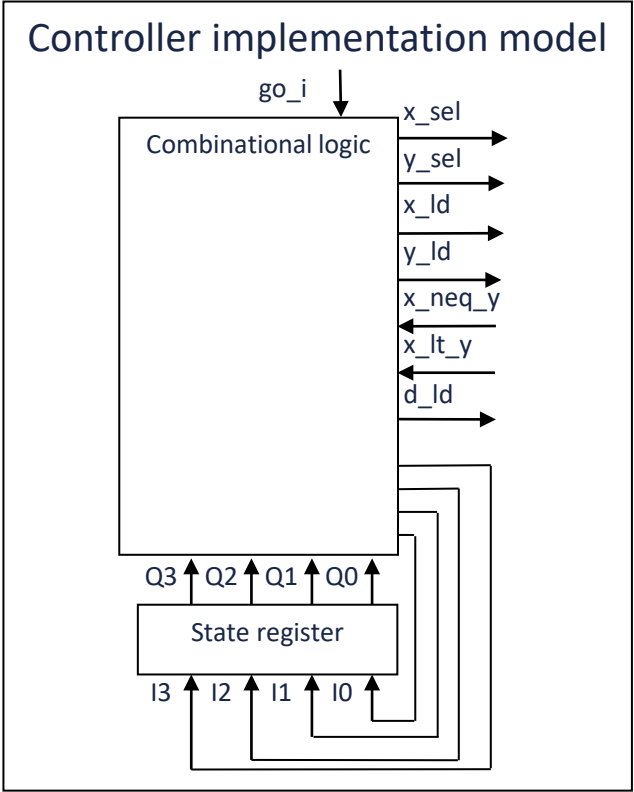
Creating the Controller's FSM



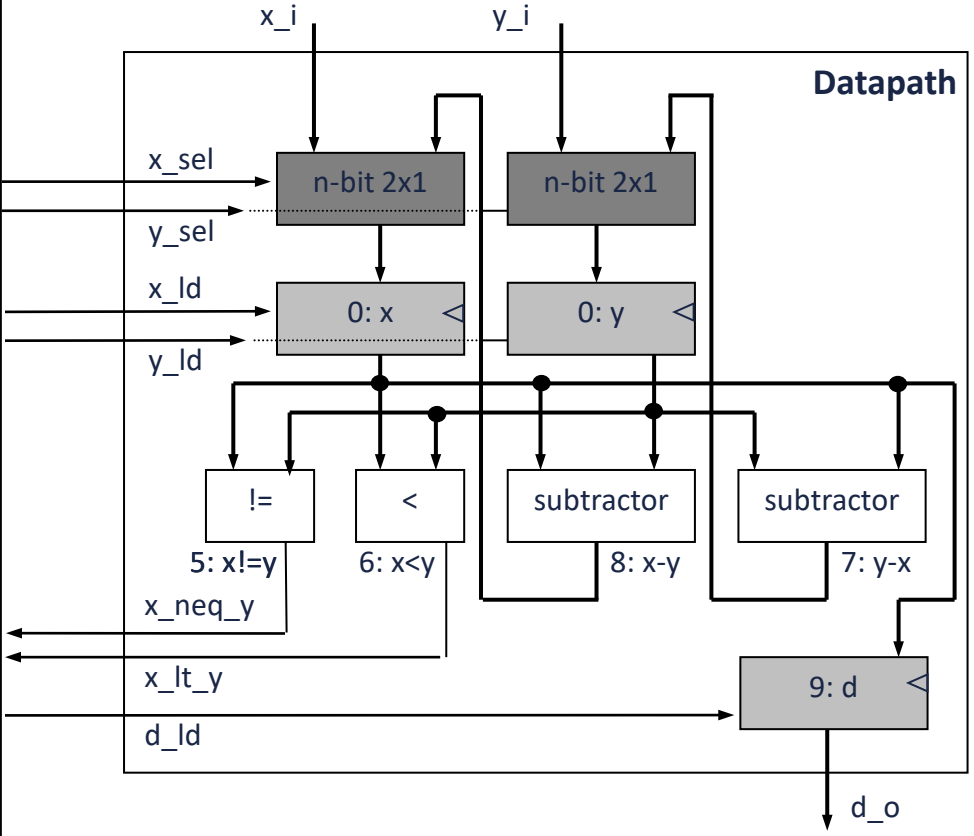
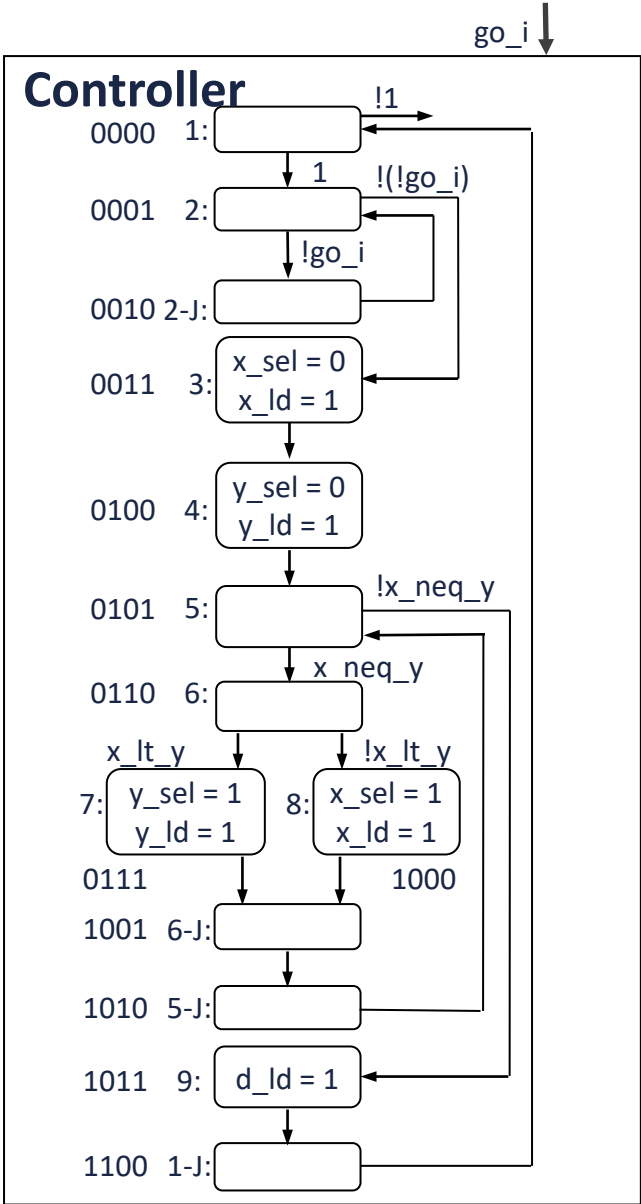
- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations



Splitting into a Controller and Datapath

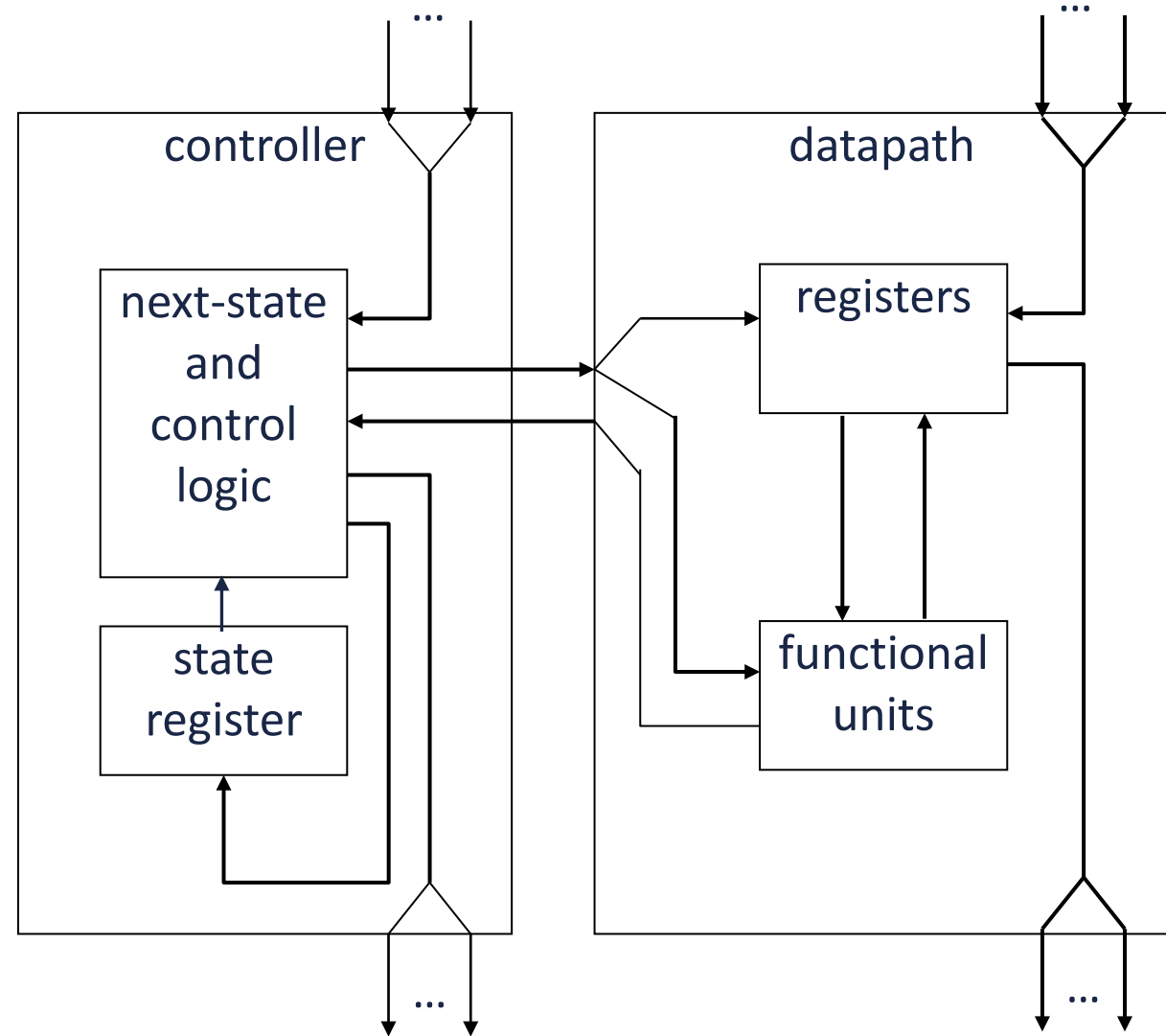


Block View of the Controller



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

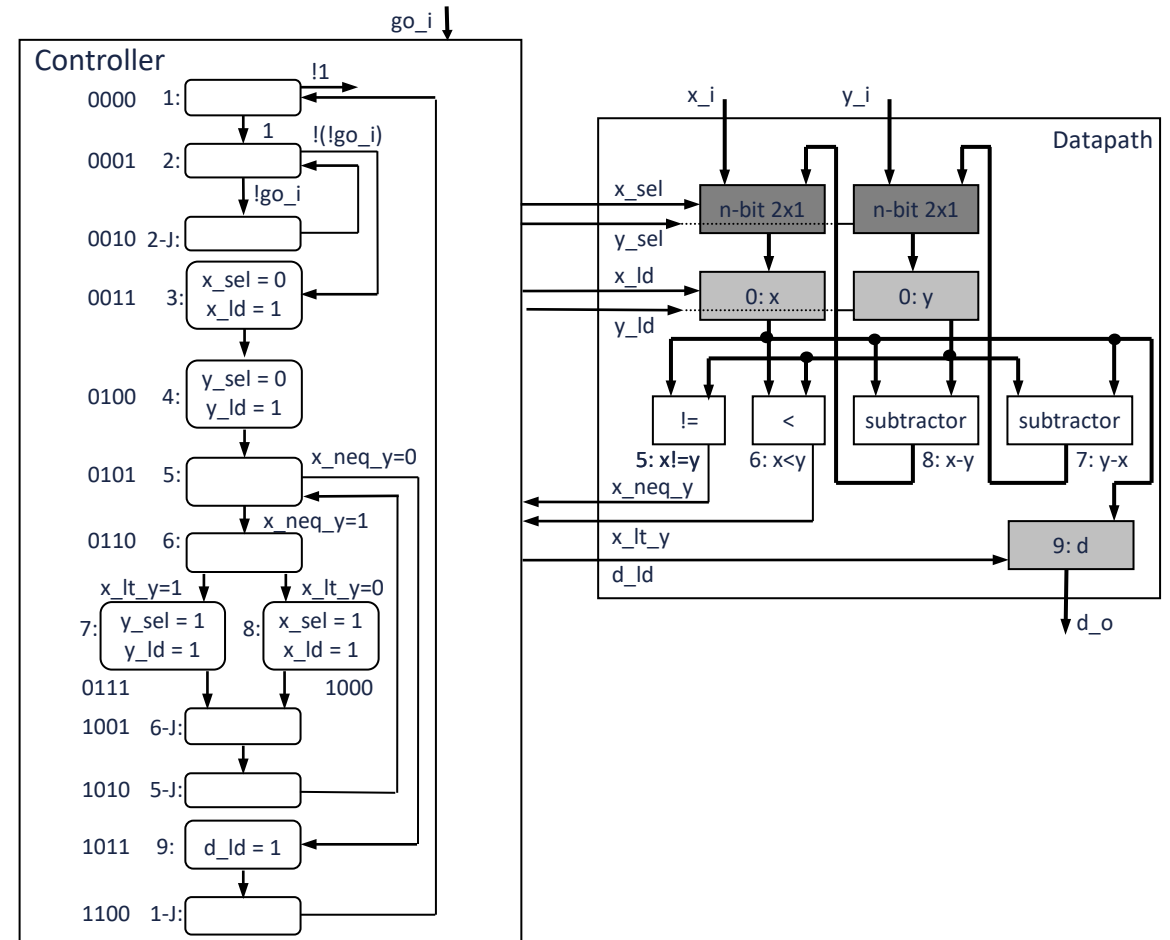
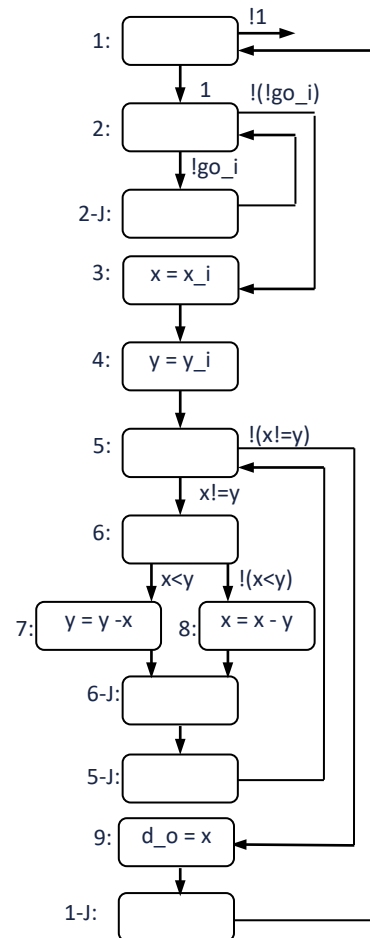
Optimizing Single-Purpose Processors

- 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;
8:     else
7:       x = x - y;
9:   }
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
7:       x = x - y;
9:   }
10:  d_o = x;
11: }
```

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)

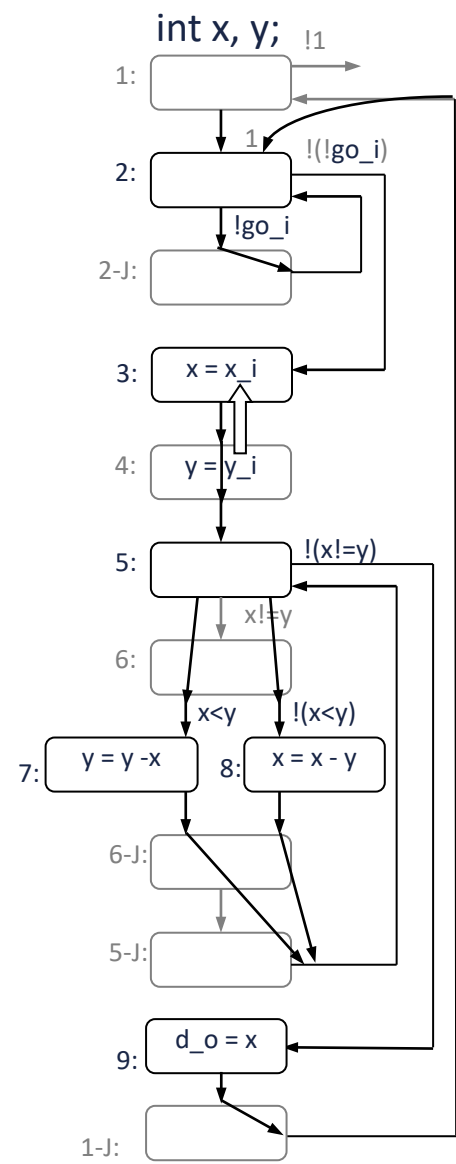
Optimization II: Optimizing the FSMD

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

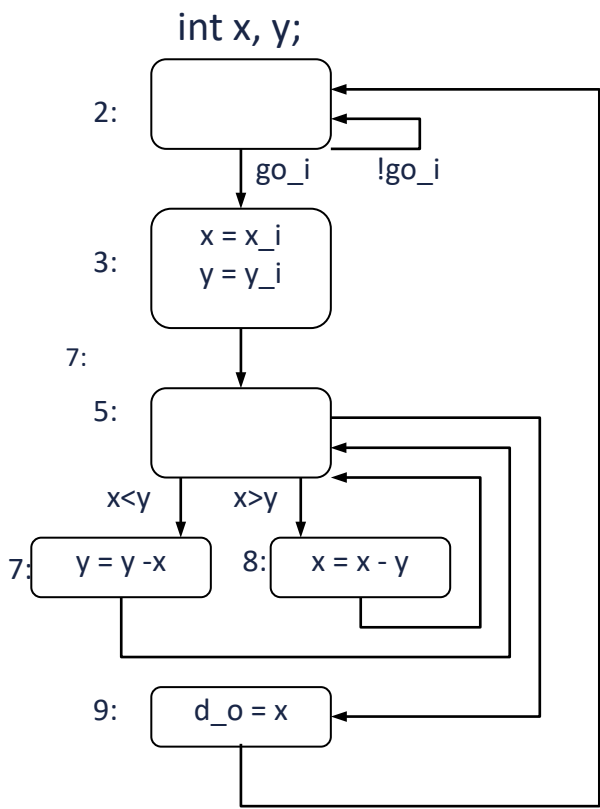
merge state 3 and state 4 – assignment operations are independent of one another

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

