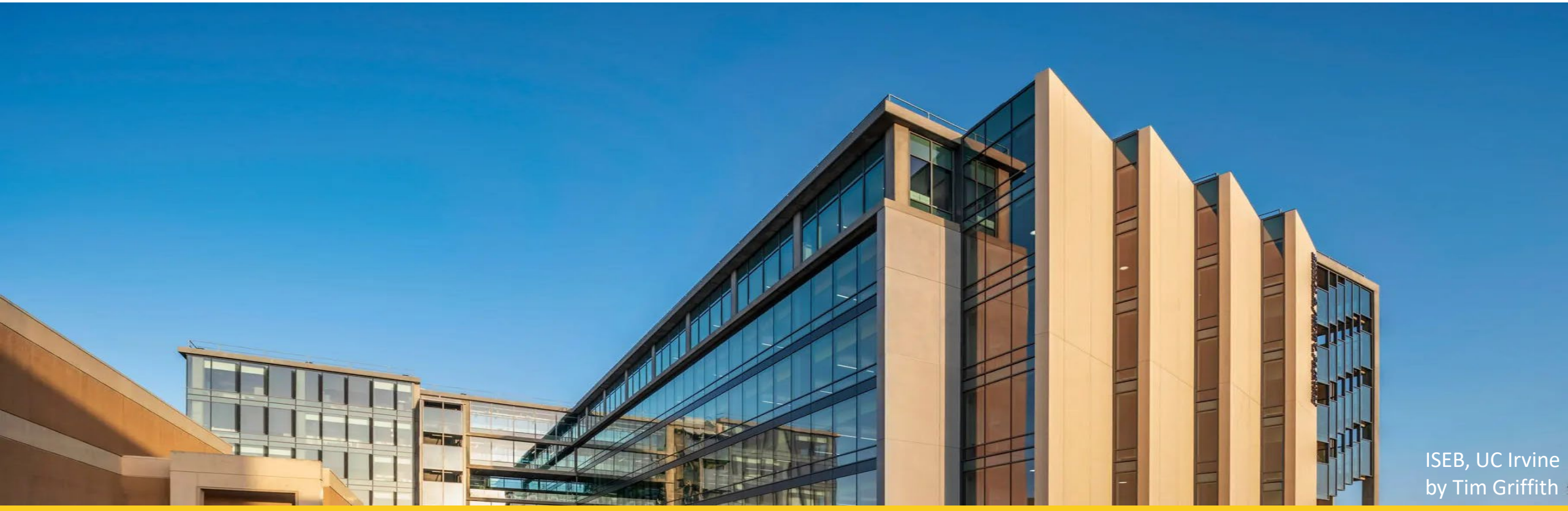
Lecture 2:

Hardware Design

Sitao Huang, sitaoh@uci.edu

October 9, 2023



ISEB, UC Irvine
by Tim Griffith

Courses I teach

UCI EECS 112: Organization of Digital Computers

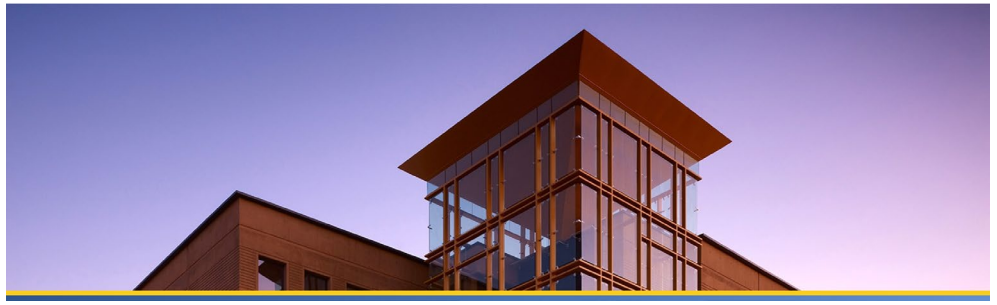
EECS 112: Organization of Digital Computers

Sitao Huang
sitaoh@uci.edu

UCI EECS 221: *Languages and Compilers for Hardware Accelerators*

EECS 221:
Languages and Compilers for Hardware Accelerators

Sitao Huang
sitaoh@uci.edu



UCI ECPS 209: CPS Case Studies

NEW

ECPS 209:
CPS Case Studies – GPU Parallel Programming

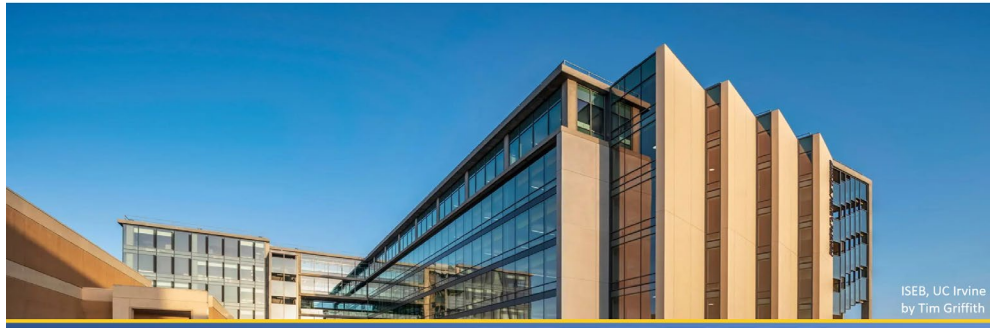
Sitao Huang
sitaoh@uci.edu

UCI EECS 298: System-on-Chip Design

NEW

EECS 298: System-on-Chip Design

Sitao Huang
sitaoh@uci.edu



AMD Xilinx FPGA Design Tools

Vitis Unified Software Platform

- **Option 1**: UCI Engineering Remote Labs:
  - Remote Windows machines
  - https://laptops.eng.uci.edu/computer-labs/remote-labs
- **Option 2**: UCI EECS Instructional Servers
  - Servers: laguna.eecs.uci.edu, bondi.eecs.uci.edu, crystalcove.eecs.uci.edu
  - Setup X11 display server (Xming for Windows and XQuartz for macOS)
  - SSH to any of these servers with your UCI credentials
    - `ssh -X yourUCInetID@laguna.uci.edu`
  - Xilinx tools installed under: `/ecelib/eceware/xilinx_2022.2/`
    - Initialization: `source /ecelib/eceware/xilinx_2022.2/Vitis/2022.2/settings64.csh`
    - Start Vitis HLS: `vitis_hls &`
- **Option 3**: Install Vitis tools on your own computer (>100 GB)
  - https://www.xilinx.com/support/download.html

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

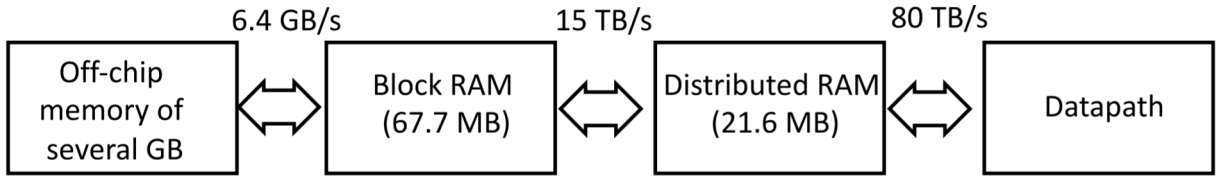
A few examples of choices in hardware accelerator design

- 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.

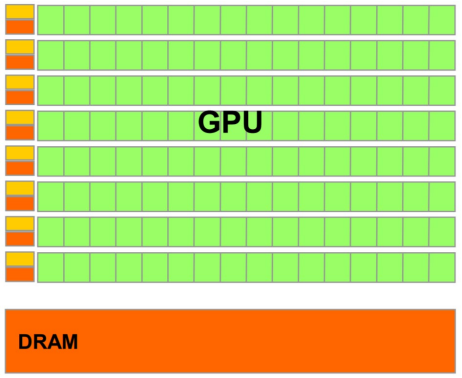
- Throughput
  - The number of tasks or results produced per unit of time
  - Memory bandwidth: how much data is moved through memory interface per unit time. MB/s or GB/s
  - Computational throughput: how much computation is done per unit time. FLOP/s, OP/s, IOP/s
  - Applications that need high throughput: image/video post-processing, DNN training, etc.

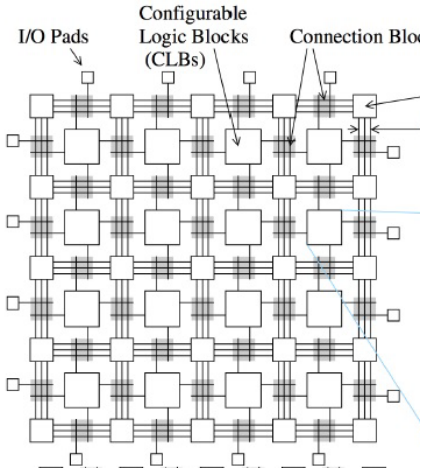


Bandwidth/memory distribution in Xilinx Virtex-7 FPGA
(source: F. Siddiqui et al, "FPGA-Based Processor Acceleration for Image Processing Applications")

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 = K
  - *Can we really achieve 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

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

- *How much memory bandwidth we need for supplying input data to K-wide adder? (not considering writes; assume adds are single cycle)*
  - *8*K bytes per clock cycle*

Parallel Hardware Design

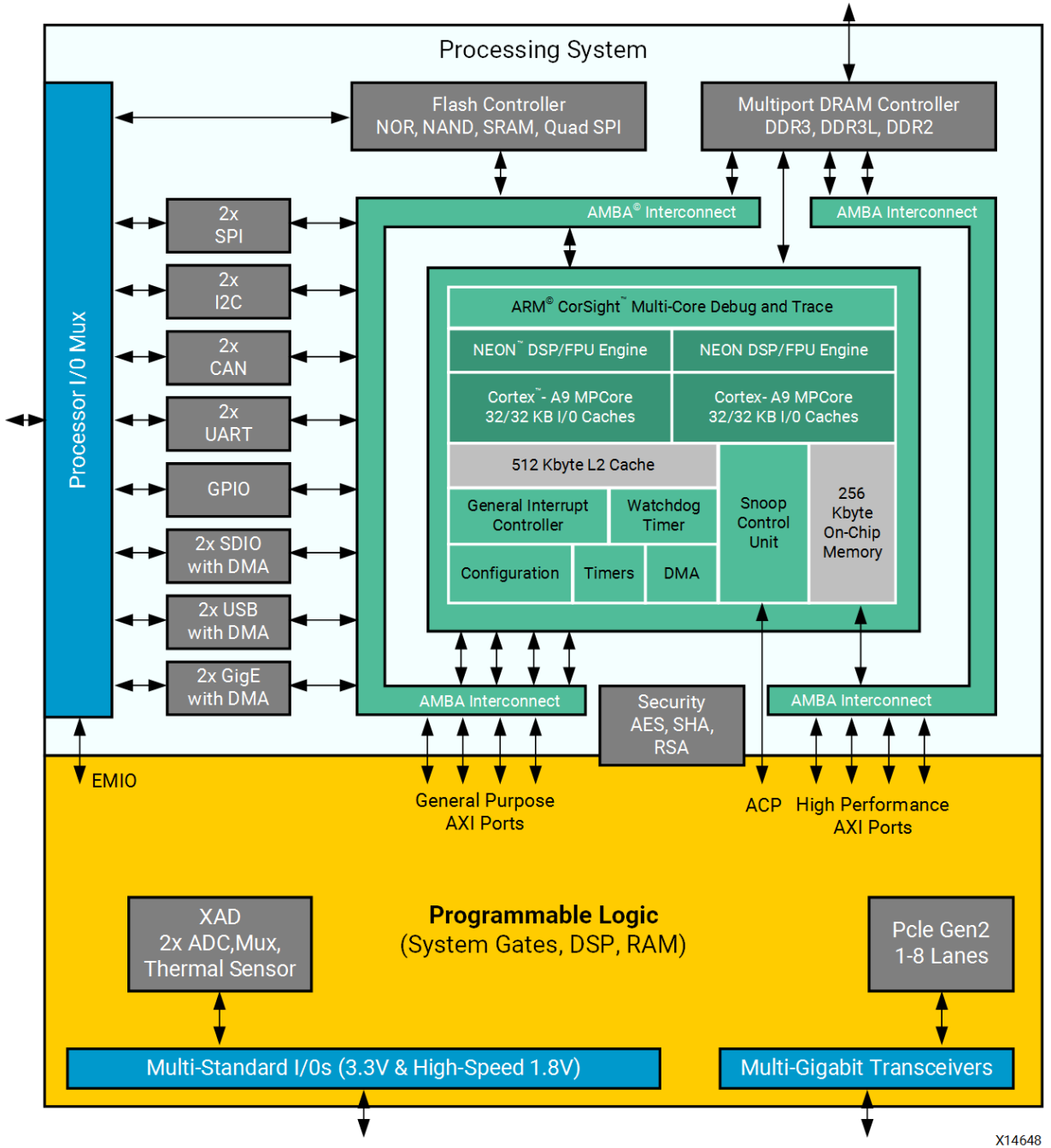
- For a specific platform, an application can be *Compute Bound* or *Memory Bound*
  - "Compute" to "Memory" ratio:

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

- 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

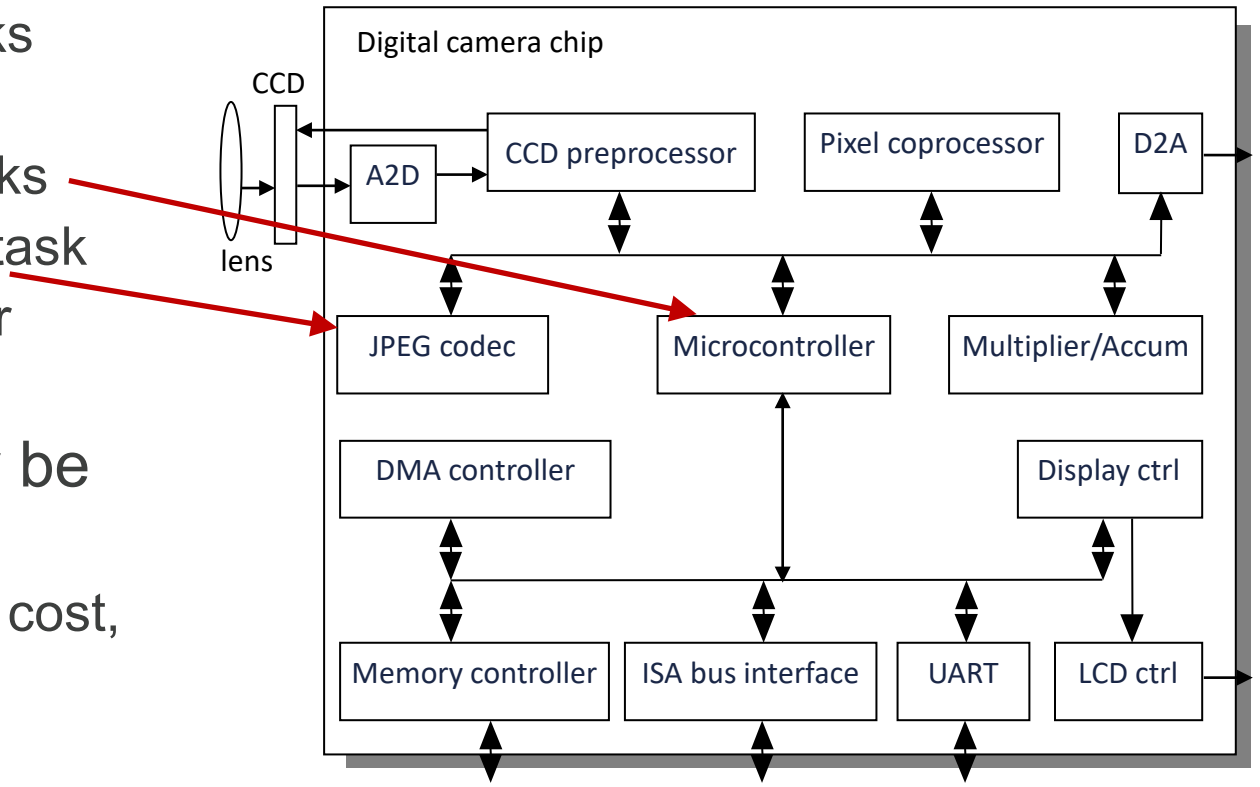
- How do data move in and out of the accelerator?
- What are the bandwidths needed for the interfaces?



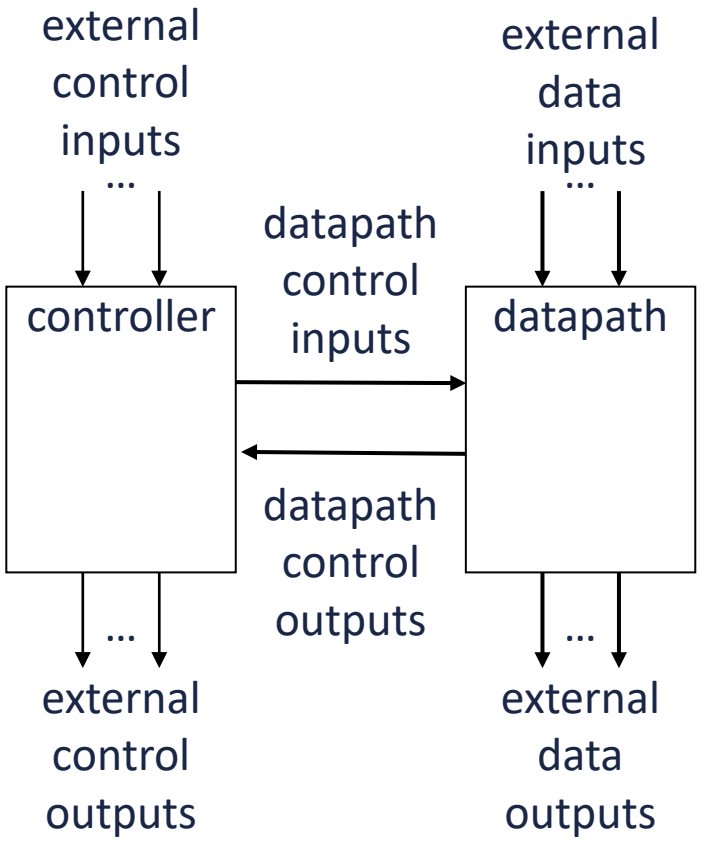
Block Diagram of AMD Xilinx Zynq-7000 SoC

Designing Single-Purpose Processors

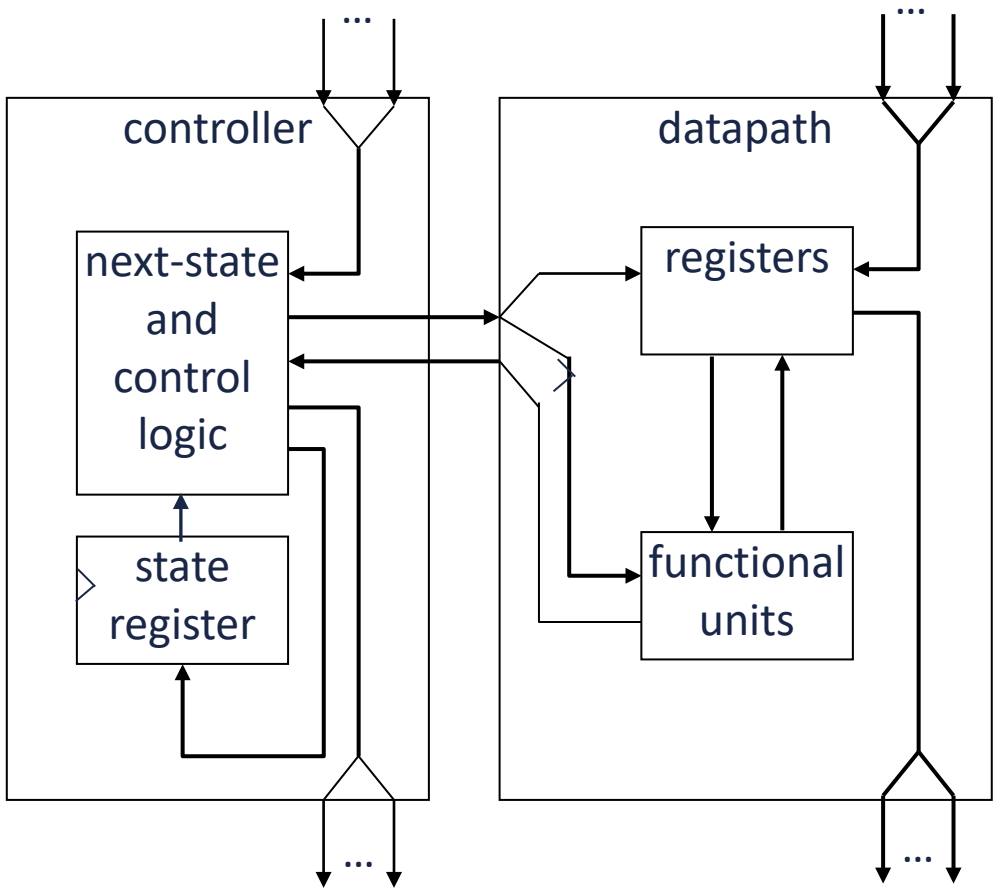
- Processor
  - Digital circuit that performs computation tasks
  - Contains controller and datapath
  - General-purpose: variety of computation tasks
  - Single-purpose: one particular computation task
  - Application-specific instruction-set processor (ASIP): domain specific tasks
- A custom single-purpose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

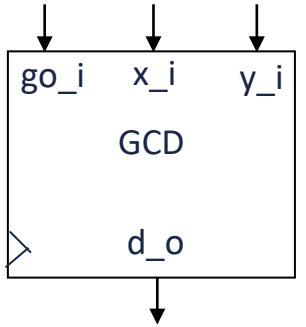


a view inside the controller and datapath

Example: Greatest Common Divisor (GCD)

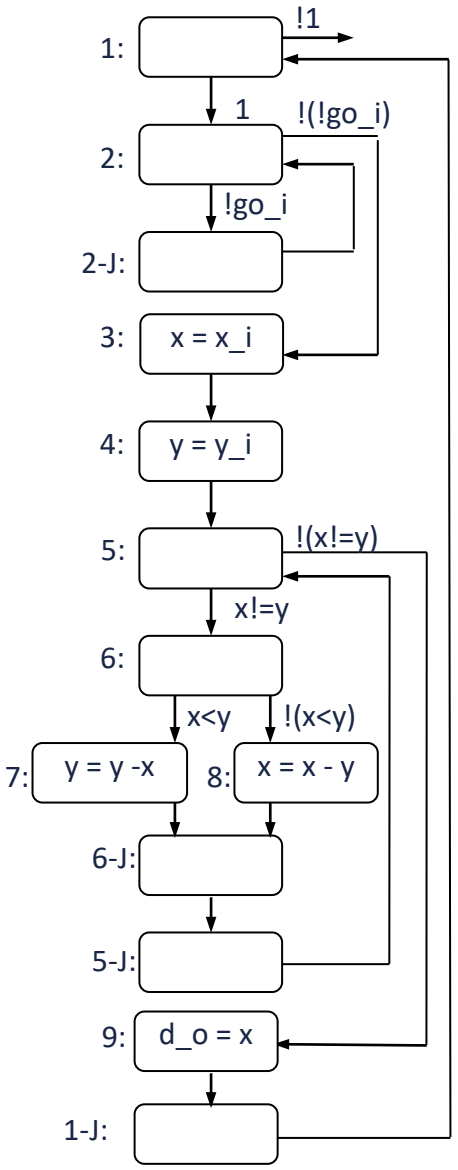
- First, create algorithm
- Then, convert algorithm to "complex" state machine
  - Known as FSMD: Finite-State Machine with Datapath
  - Can use templates to perform such conversion

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

(b) desired functionality

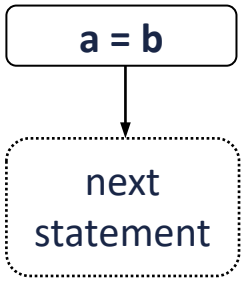


(c) FSMD

State Diagram Templates

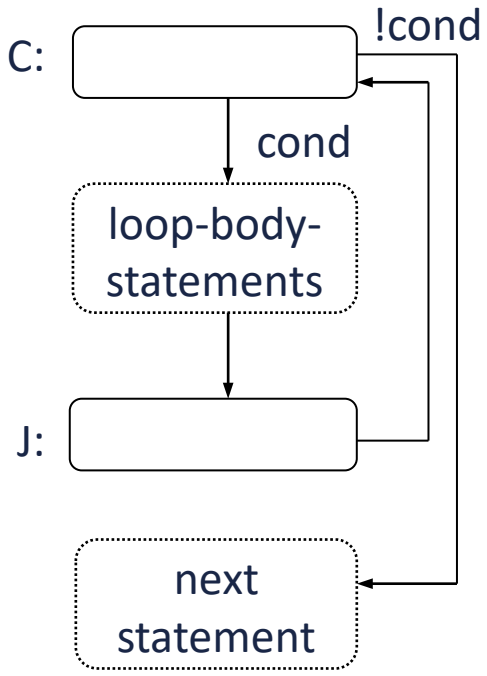
Assignment statement

a = b
next statement



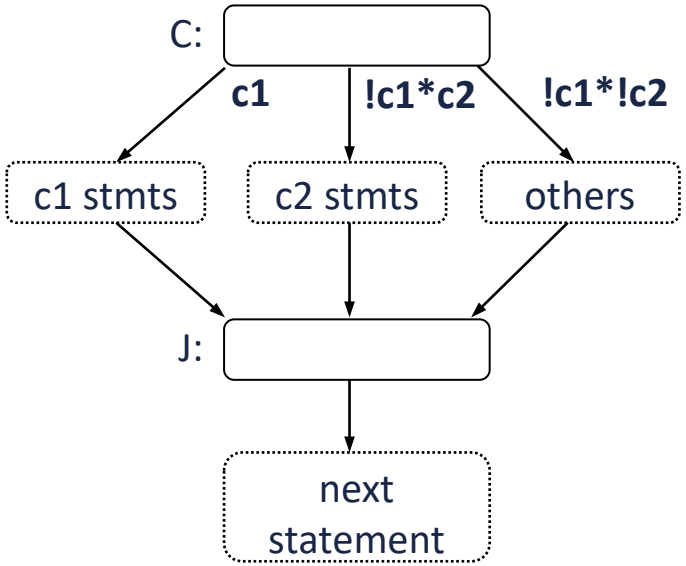
Loop statement

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



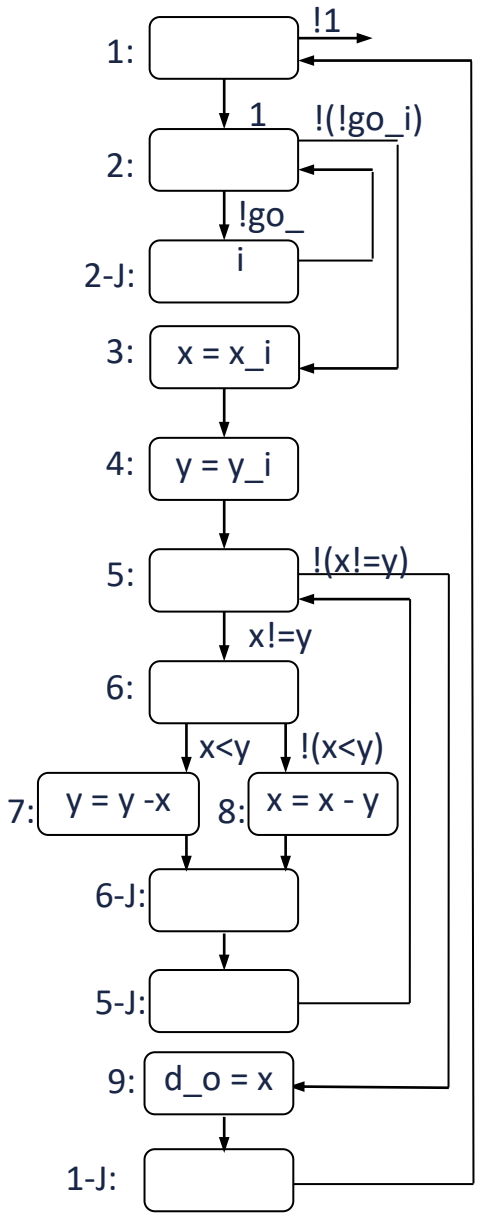
Branch statement

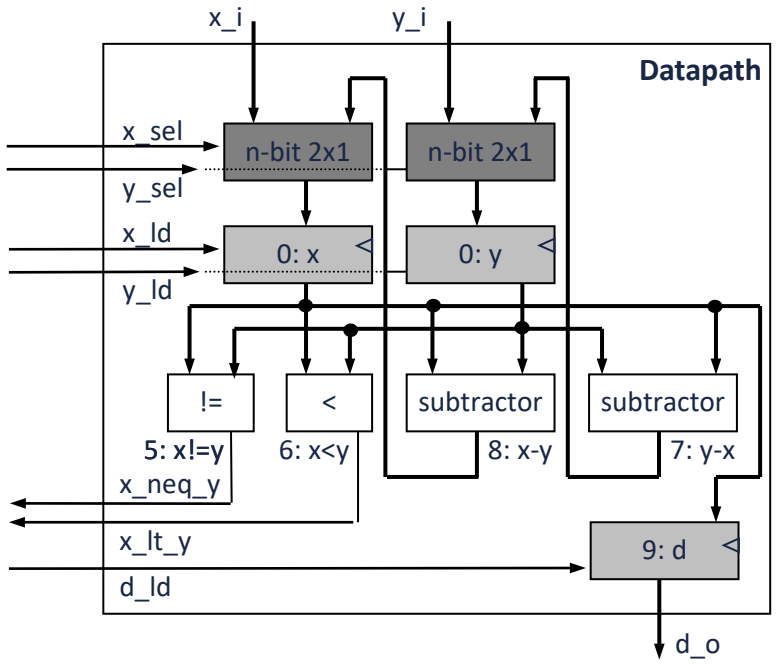
if (c1)
    c1 stmts
else if c2
    c2 stmts
else
    other stmts
next statement



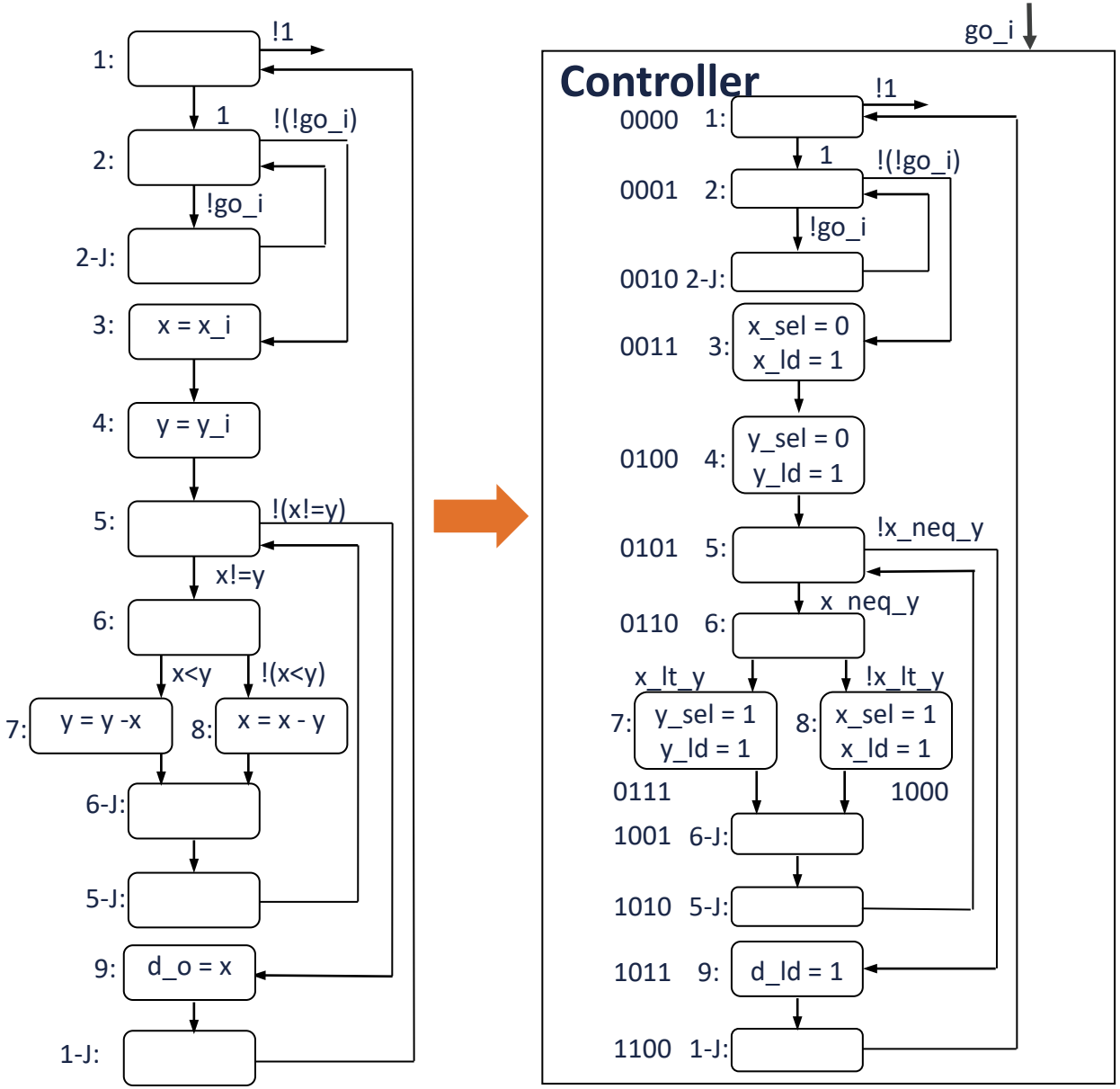
Creating the Datapath

- 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

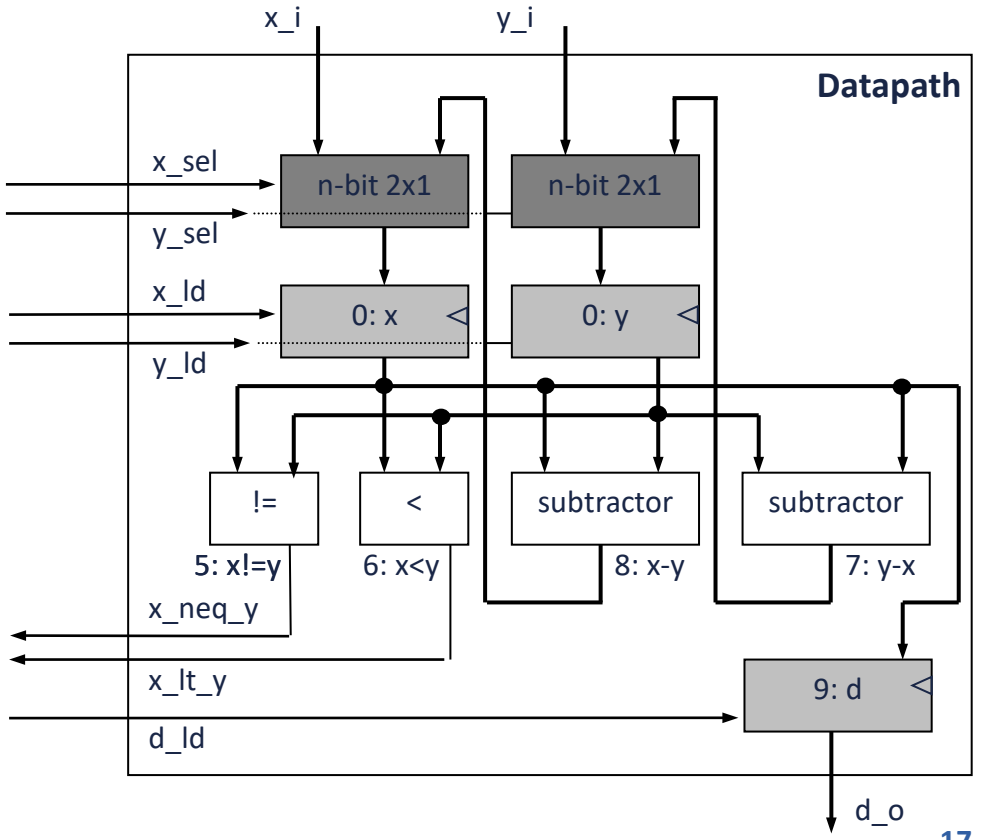




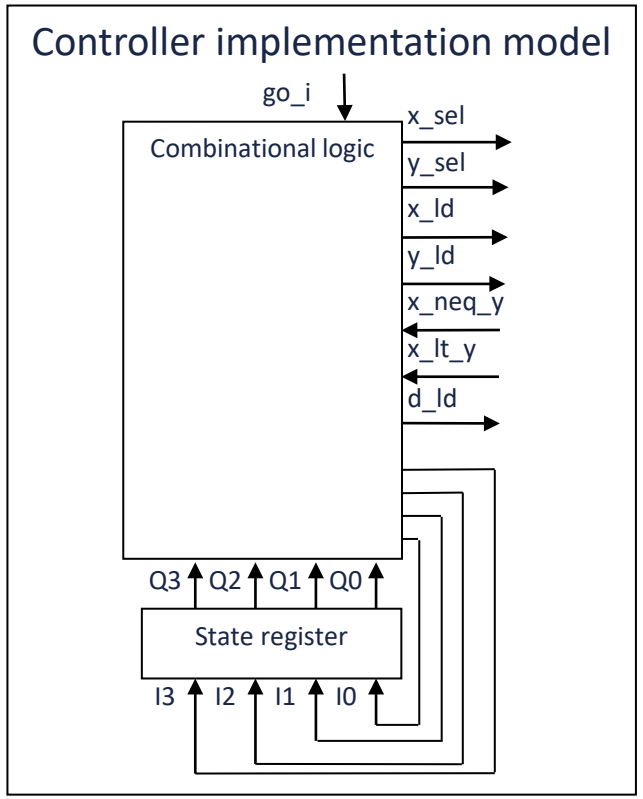
Creating the Controller's FSM



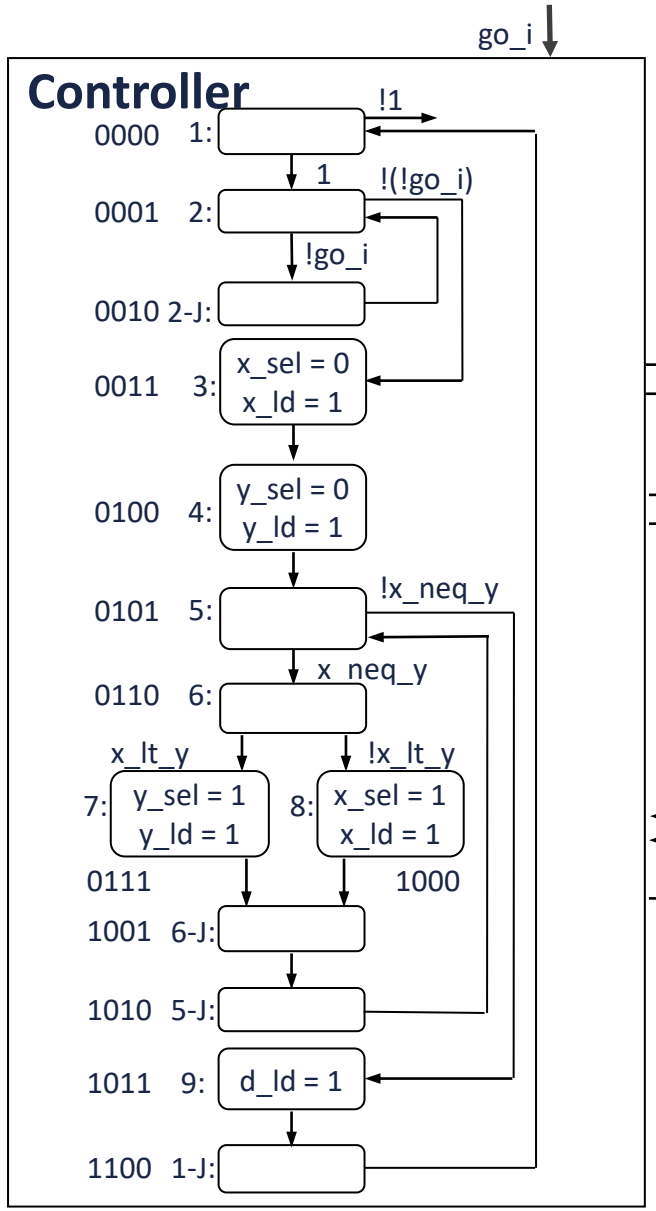
- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations

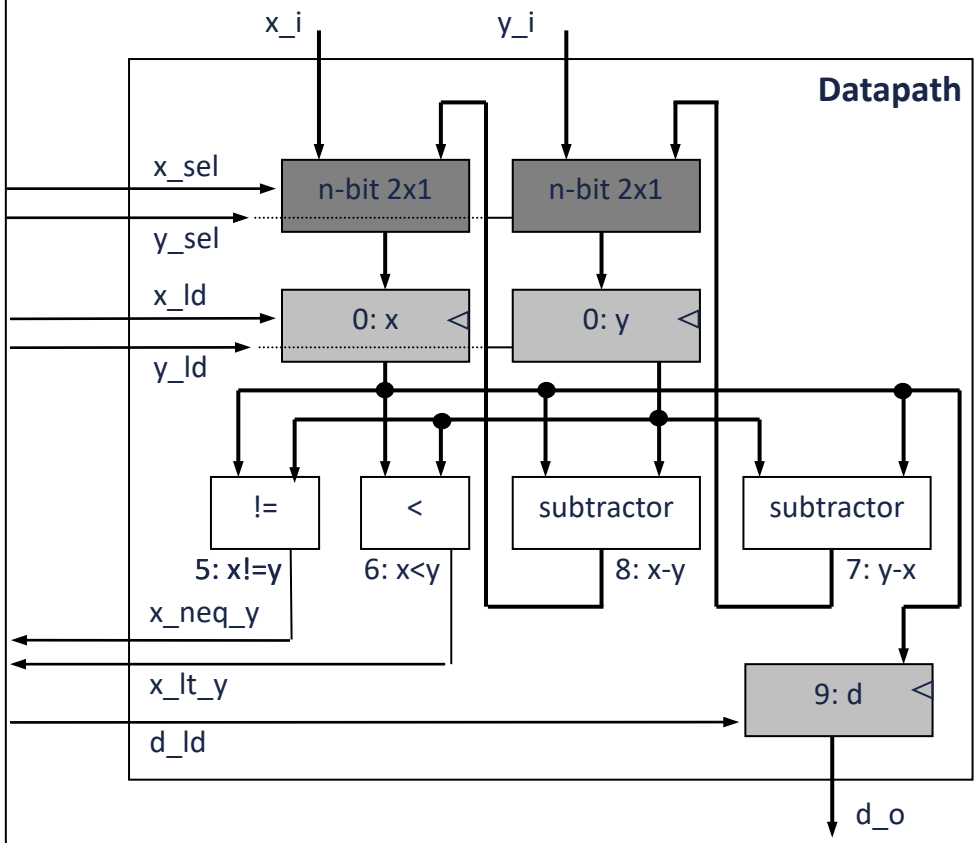


Splitting into a Controller and Datapath



Block View of the Controller

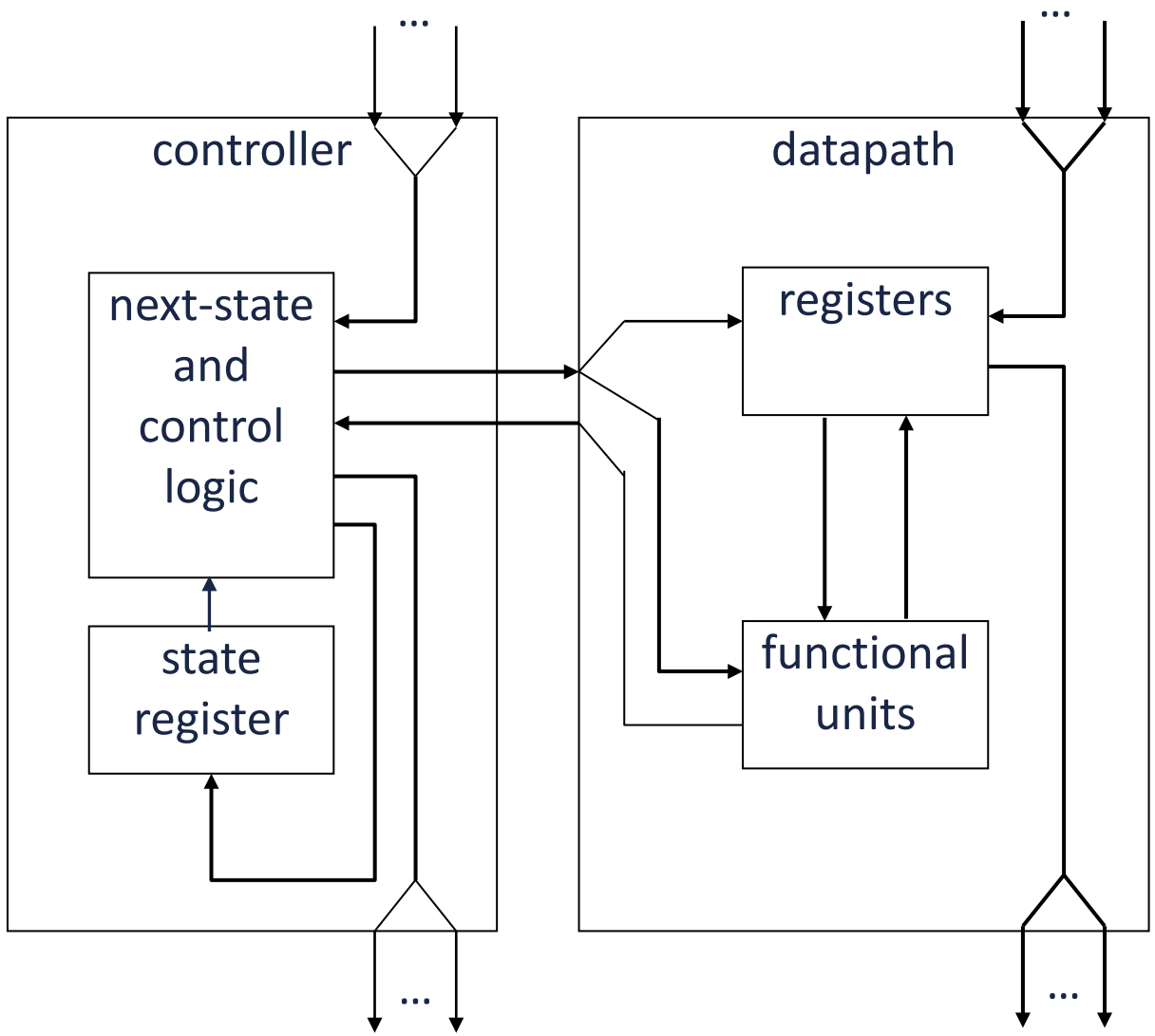




Completing the GCD Single-Purpose Processor Design

- 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*

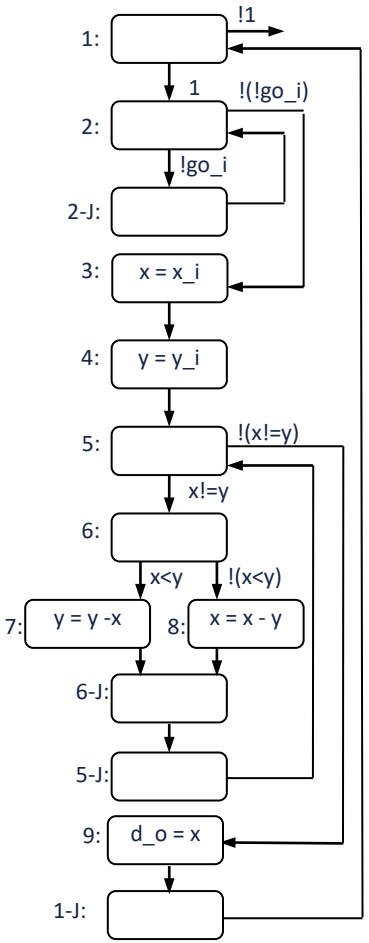


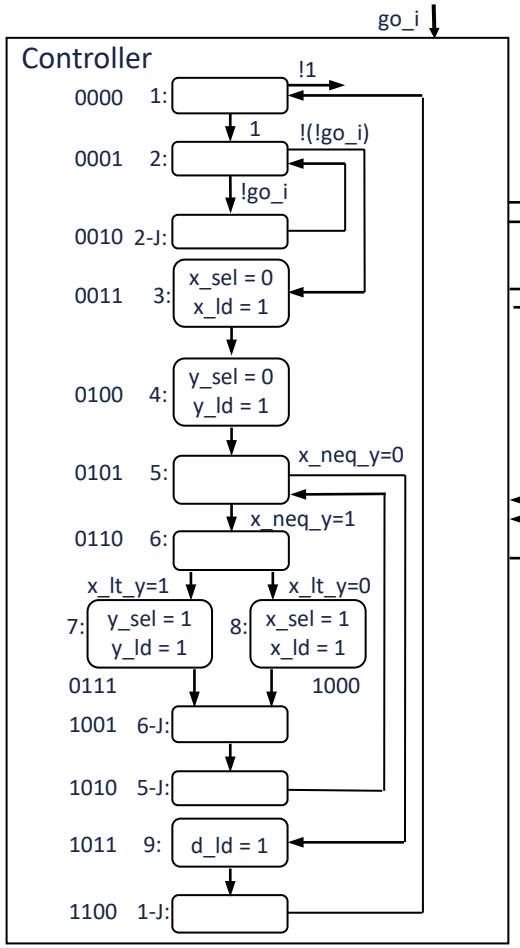
a view inside the controller and datapath

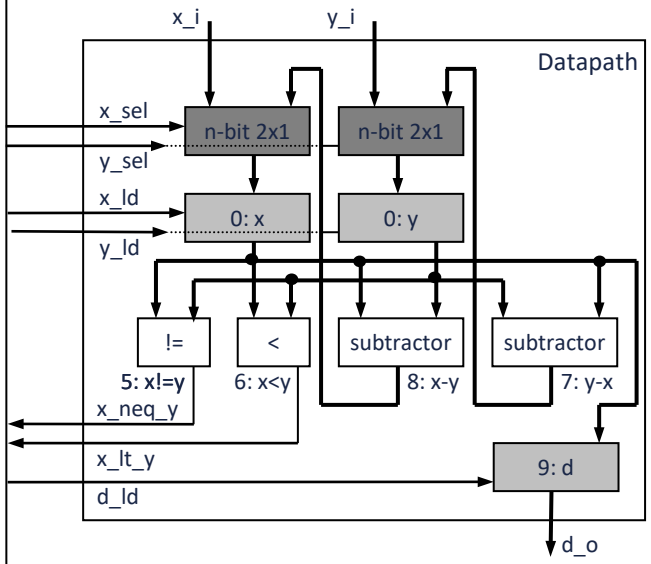
Optimizing Single-Purpose Processors

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

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

optimized program

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

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)

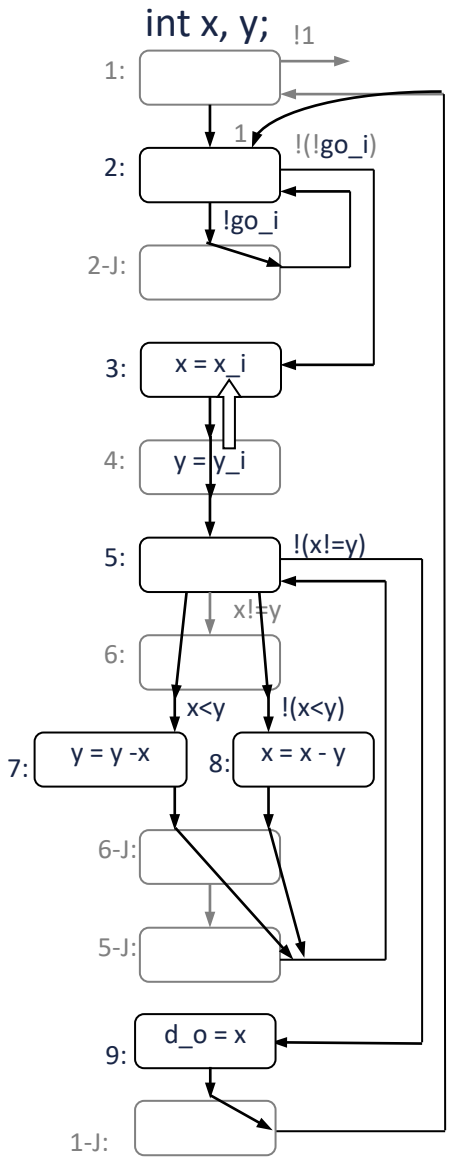
Optimization II: Optimizing the FSMD

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

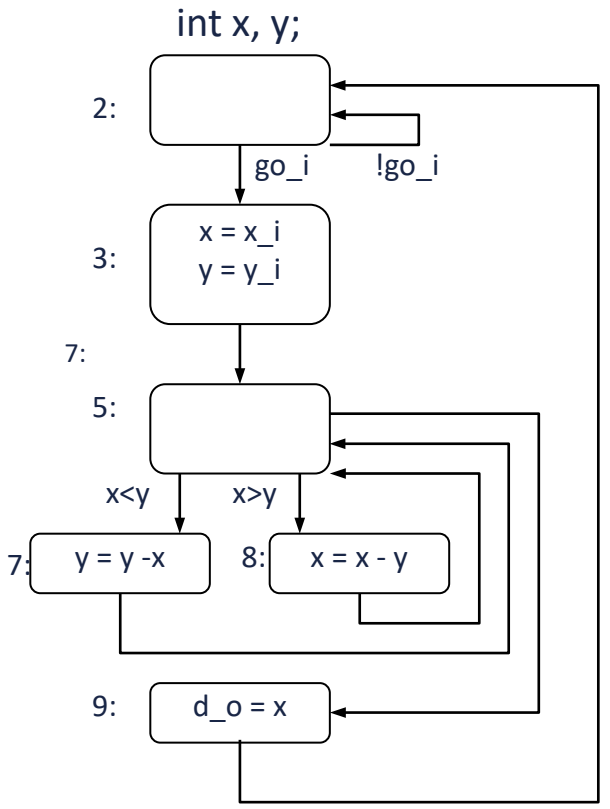
merge state 3 and state 4 – assignment operations are independent of one another

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

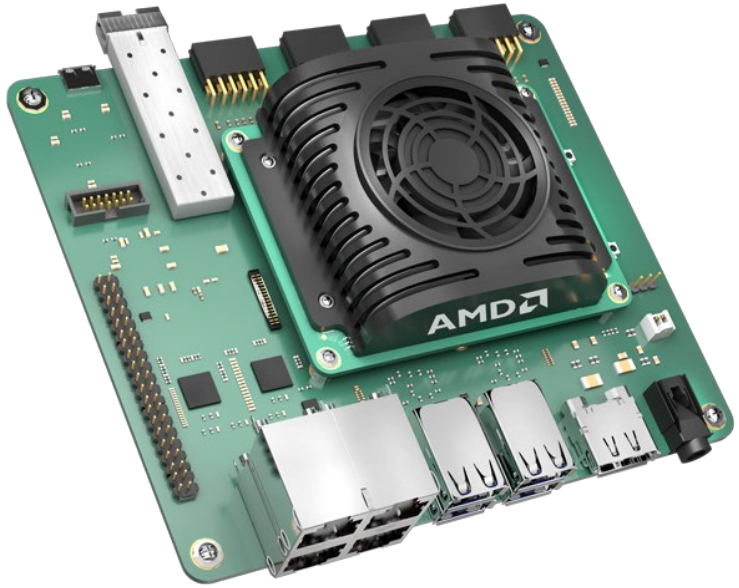
- Sharing of functional units
  - One-to-one mapping, as done previously, is not necessary
  - If same operation occurs in different states, they can share a single functional unit

- 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

Experiment Platform: AMD Kria KR260 Robotics Starter Kit



- Development platform with Kria K26 SOM (System on Module)
- Built for robotics and industrial applications
- Native ROS (Robot Operating System) 2 support
- Chip: Zynq™ UltraScale+™ MPSoC EV (XCK26)

KR260 Power Supply & Adapter
- 12V 36W

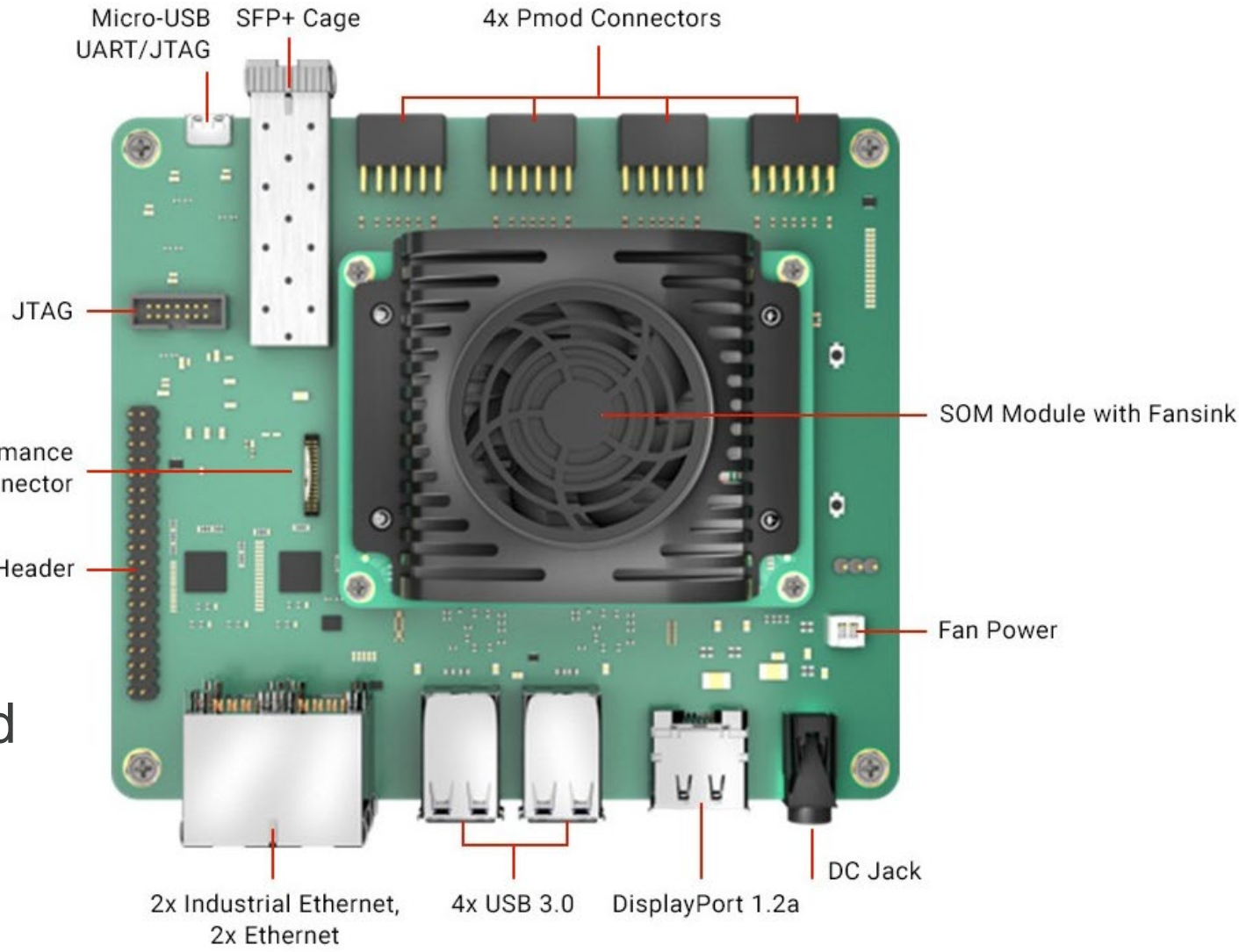
MicroSD & Adapter
- 64 GB
- System image to boot

USB-A to micro-B Cable
- Establish UART connection to setup terminal

Ethernet Cable
- Networking
- PYNQ , Jupyter notebook

Experiment Platform: AMD Kria KR260 Robotics Starter Kit

- Device: Zynq™ UltraScale+™ MPSoC EV (XCK26)
- System logic cells: 256K
- 36Kb Block RAM blocks: 144
- 288Kb UltraRAM blocks: 64
- DSP slices: 1.2K
- DDR memory: 4GB DDR4
- Primary boot memory: 512 Mb QSPI
- Secondary boot memory: SDHC card



EECS 298:
System-on-Chip Design

Sitao Huang, sitaoh@uci.edu