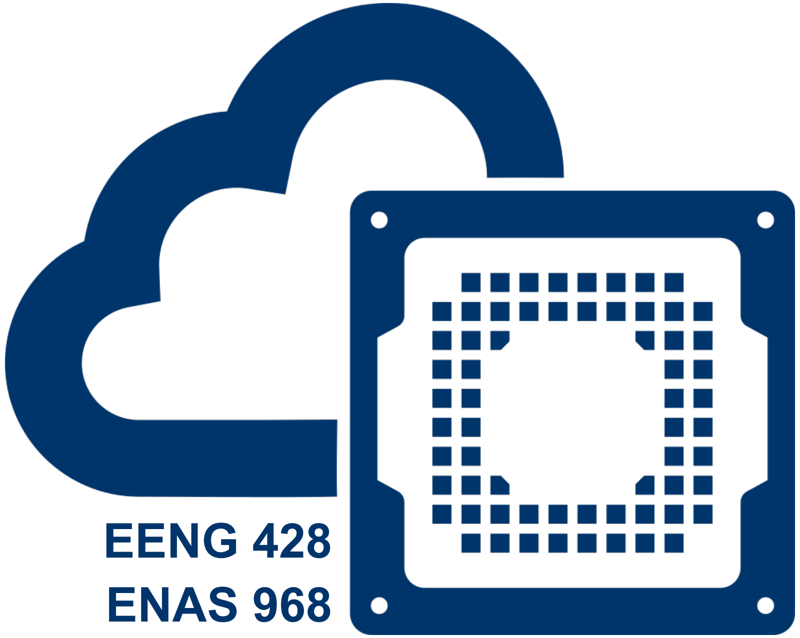
Cloud FPGA



bit.ly/cloudfpga

Lecture: Amazon F1 HDK and SDK

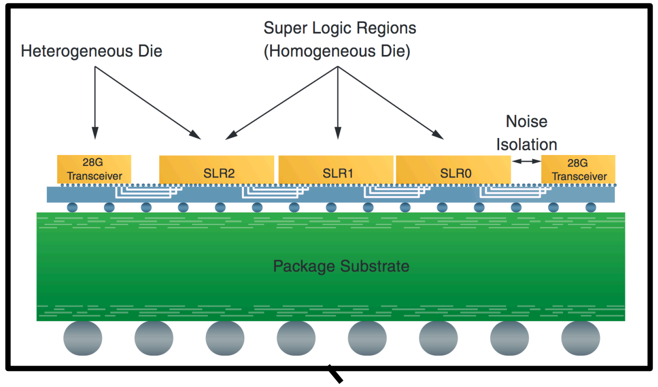
Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

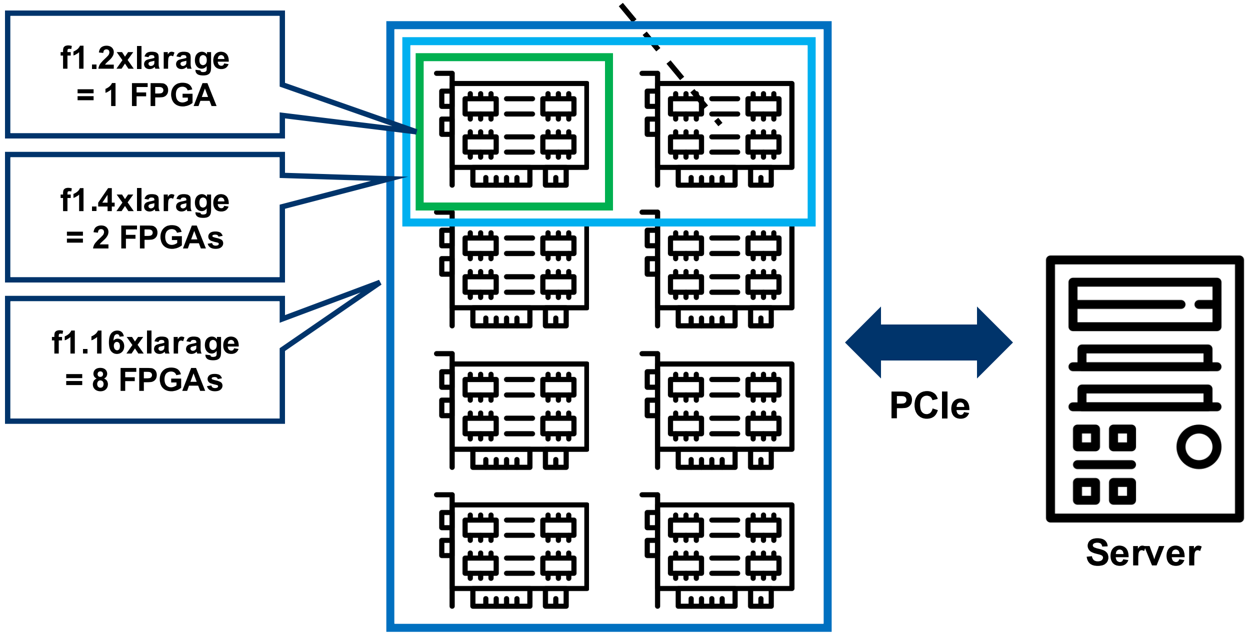
EENG 428 / ENAS 968
Cloud FPGA

Amazon F1 Cloud FPGAs

- Cloud FPGAs in Amazon Web Services let users access large numbers of UltraScale+ FPGAs in different geographic regions
- Amazon Web Services provides access to FPGAs via a **f1.2xlarge**, **f1.4xlarge**, and **f1.16xlarage** instances
  - Actual configuration is not publicly known, but can assume 8 FPGA servers where smaller number of FPGAs are given to each user
  - The 2x, 4x, and 16x can share all servers or there may be dedicated servers for each instance type
- Amazon provides tools for programming the FPGA and software development for using the hardware running on FPGAs with the server: the HDK and SDK

Xilinx Virtex UltraScale+





f1.2xlarage
= 1 FPGA

f1.4xlarage
= 2 FPGAs

f1.16xlarage
= 8 FPGAs

PCIe

Server

FPGA boards

Amazon F1 Cloud FPGAs

Running your hardware in Cloud FPGA requires two main components:

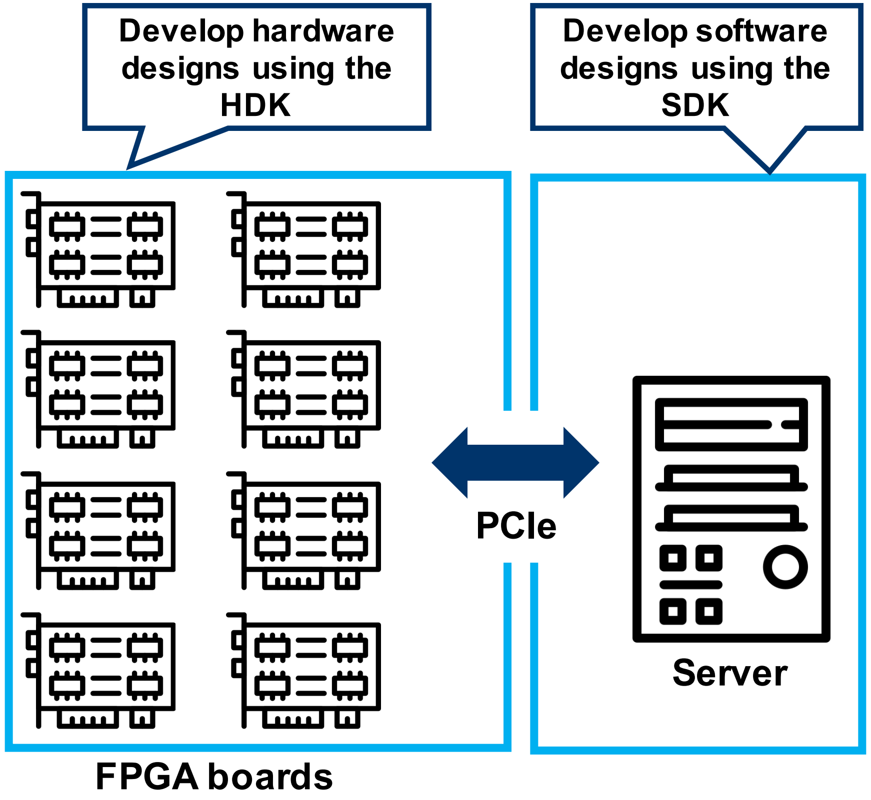
- Hardware design loaded on the FPGAs
- Software running on the server to communicate with the hardware design that is on the FPGA

Hardware Development Kit (HDK):

- Develop the design and create bitstreams, also called Amazon FPGA Images (AFIs)
  - No need to use HDK if using pre-built AFI

Software Development Kit (SDK):

- Tools for High-Level Synthesis (not needed if developing your own Verilog code)
- Tools for loading AFIs and interacting with FPGAs
- C libraries and Python bindings, plus Linux drivers for software that uses the FPGAs



HDK and SKD git: https://github.com/aws/aws-fpga

Development and Deployment Process in AWS

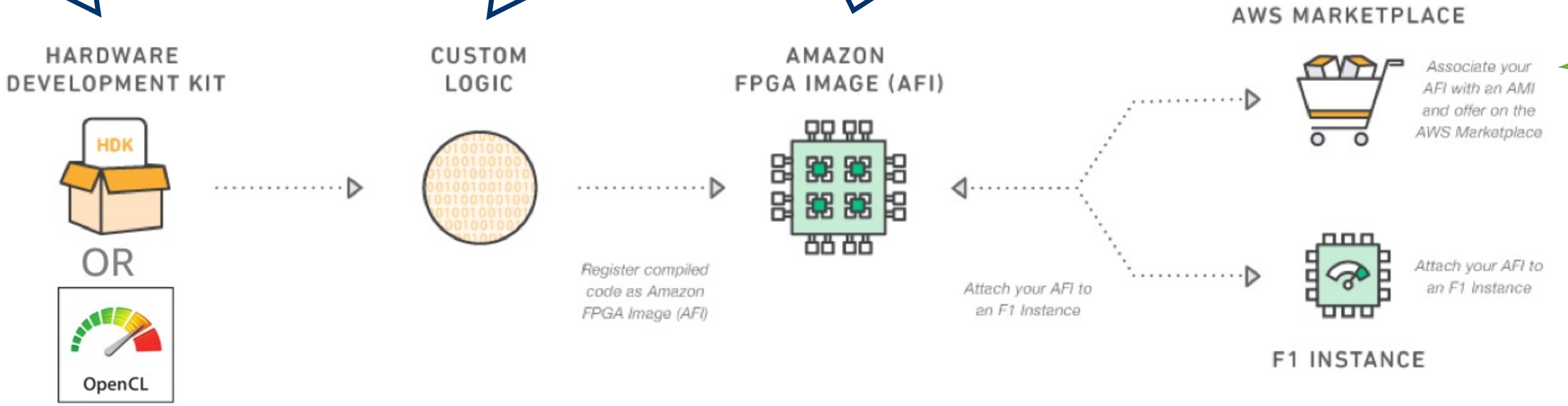
Development of F1 hardware, and using the hardware, has four steps

Develop the hardware, in Verilog (or other HDL), or using higher-level languages like OpenCL, or using HLS

Compile (synthesize) the design into the custom hardware module

Get the design approved and obtain an Amazon FPGA Image

Rent F1 instance to run the design, or give (or sell) access to the AFI to others



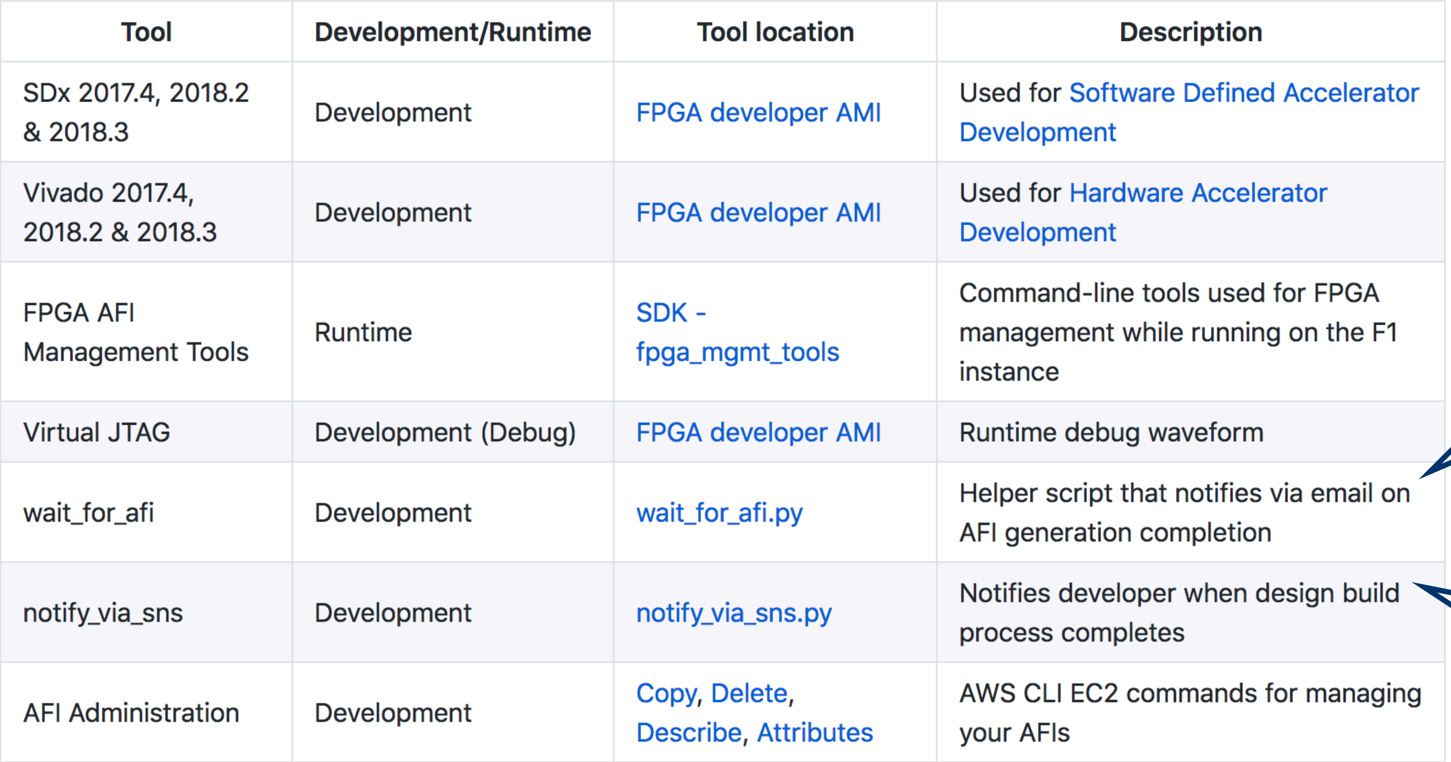
Allows to sell access to hardware designs

- Development is enabled by the HDK and SDK, and deployment is enabled by EC2 and the AWS marketplace

Image from:
https://github.com/aws/aws-fpga/

Overview of Development Tools

- A number of tools are provided to aid in the development
  - Mostly Xilinx tools
  - Plus scripts and custom IP cores
  - May examples of custom logic are also provided



Tool	Development/Runtime	Tool location	Description
SDx 2017.4, 2018.2 & 2018.3	Development	FPGA developer AMI	Used for Software Defined Accelerator Development
Vivado 2017.4, 2018.2 & 2018.3	Development	FPGA developer AMI	Used for Hardware Accelerator Development
FPGA AFI Management Tools	Runtime	SDK - fpga_mgmt_tools	Command-line tools used for FPGA management while running on the F1 instance
Virtual JTAG	Development (Debug)	FPGA developer AMI	Runtime debug waveform
wait_for_afi	Development	wait_for_afi.py	Helper script that notifies via email on AFI generation completion
notify_via_sns	Development	notify_via_sns.py	Notifies developer when design build process completes
AFI Administration	Development	Copy, Delete, Describe, Attributes	AWS CLI EC2 commands for managing your AFIs

Designs are always submitted to Amazon for approval and to get AFI, seems mostly automated process now, but still can take some time

Build process can take many hours; also want to shut down VM instance once build finishes to save money

Share:
bit.ly/cloudfpga

Information from:
https://github.com/aws/aws-fpga/

Amazon F1 HDK

Hardware Development Kit

Amazon’s FPGA **Hardware Development Kit (HDK)**:

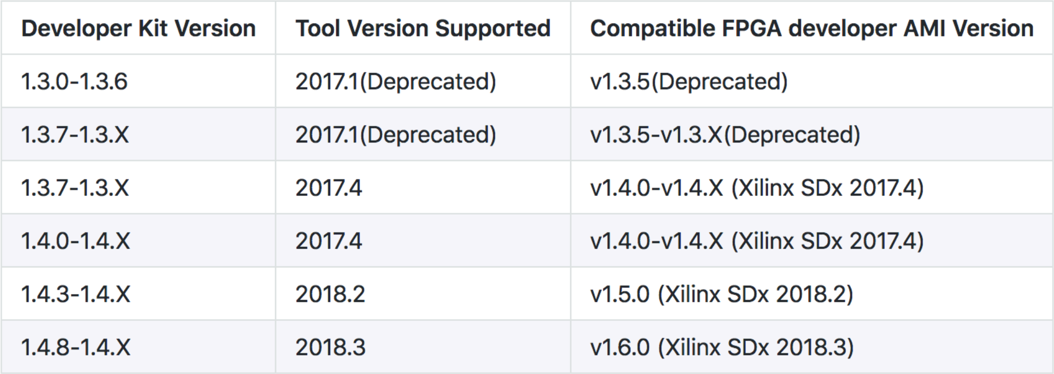
- Contains useful information, examples, and scripts for building hardware designs and generating the Amazon FPGA Images (AFI)
- Includes the development environment, simulation, build and AFI creation scripts
- It contains Xilinx’s Vivado tools, plus IP cores, e.g. PCIe, and custom scripts form Amazon
  - Can be run in “development” VM on most EC2 instances
  - Can potentially run locally on your own machine

Recall cost of different types of instances, F1 instances are expensive and FPGA is not required when developing the code:

- Develop design on instance with no FGPA, e.g. c4.4xlarge
- Later load up f1 instance to actually run the AFI on an FPGA

Need Xilinx license and IP cores corresponding to ones used by Amazon

Different HDK versions exist



Developer Kit Version	Tool Version Supported	Compatible FPGA developer AMI Version
1.3.0-1.3.6	2017.1(Deprecated)	v1.3.5(Deprecated)
1.3.7-1.3.X	2017.1(Deprecated)	v1.3.5-v1.3.X(Deprecated)
1.3.7-1.3.X	2017.4	v1.4.0-v1.4.X (Xilinx SDx 2017.4)
1.4.0-1.4.X	2017.4	v1.4.0-v1.4.X (Xilinx SDx 2017.4)
1.4.3-1.4.X	2018.2	v1.5.0 (Xilinx SDx 2018.2)
1.4.8-1.4.X	2018.3	v1.6.0 (Xilinx SDx 2018.3)

Hardware design needs to match the HDK version, may need to check out older HDK version to get the version to match

Information and HDK version table from [1]

HDK and Hardware Development Concepts

A brief list of concepts listed by Amazon relating to the development of the use of HDK and developing hardware on FPGAs:

- Scripting languages (shell, tcl)
- RTL (Verilog or VHDL) development
- Synthesis tools and the iterative process of identifying timing critical paths and optimizing hardware to meet timing
- Familiarity with concepts related to designing for FPGAs, DMA, DDR, AXI protocol and Linux drivers
- RTL simulation and experience with simulation debug or FPGA runtime waveform viewer debug methods

Most HDK commands are invoked from the Linux shell; while tcl (tool command language) scripts are used by Vivado tools

Other languages can be used such as SystemVerilog

Leverage ideas such as pipelining, parallelism, etc., covered in small part by the course and textbook

FPGA design and AXI are almost required, others are “hidden” by scripts and tools provided

Standard part of design process, but Amazon also some custom solutions like virtual JTAG, virtual LEDs, and virtual DipSwitches

Concept list taken from [1]

Developing Designs with the HDK

Development cycle can be roughly broken into three steps when using Amazon servers for all steps of the development process:

1. Develop and simulate the design
    - Write code, check for bug
    - Simulate, check design works
2. Synthesize the design
    - Make sure timing and other parameters are met
    - Submit digital checkpoint to create AFI
3. Run the design on an FPGA
    - Actually use the design!

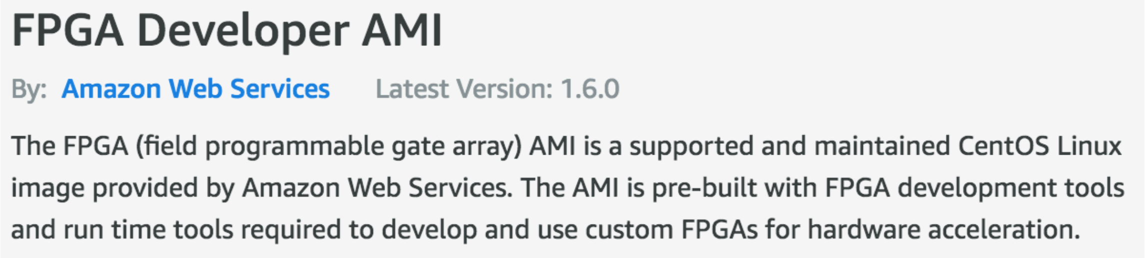
Perform on less expensive instance, e.g., **t2.2xlarge**, as does not require lot of computational power

Perform on more expensive instance, e.g., **c4.2xlarge**, to compiler faster

Perform on most expensive instance, e.g., **f1.2xlarge**, to actually use the FPGA

FPGA Developer AMI:

- VM image pre-loaded with Vivado and required licenses



FPGA Developer AMI

By: Amazon Web Services   Latest Version: 1.6.0

The FPGA (field programmable gate array) AMI is a supported and maintained CentOS Linux image provided by Amazon Web Services. The AMI is pre-built with FPGA development tools and run time tools required to develop and use custom FPGAs for hardware acceleration.

Concept list taken from [1]

Developing Designs Locally

Designs can be developed in large part locally, even without HDK:

- Write Verilog code in any editor
- Check syntax and basic debugging with testbenches and **`iverilog`**, for example
- If using standard interface such as AXI, can test whole design with testbench that also uses AXI

Regardless of approach, it is good practice to have testbench for each module anyway; don't wait with testing until whole design is done!

Can have possible issues with different AXI versions and implementation

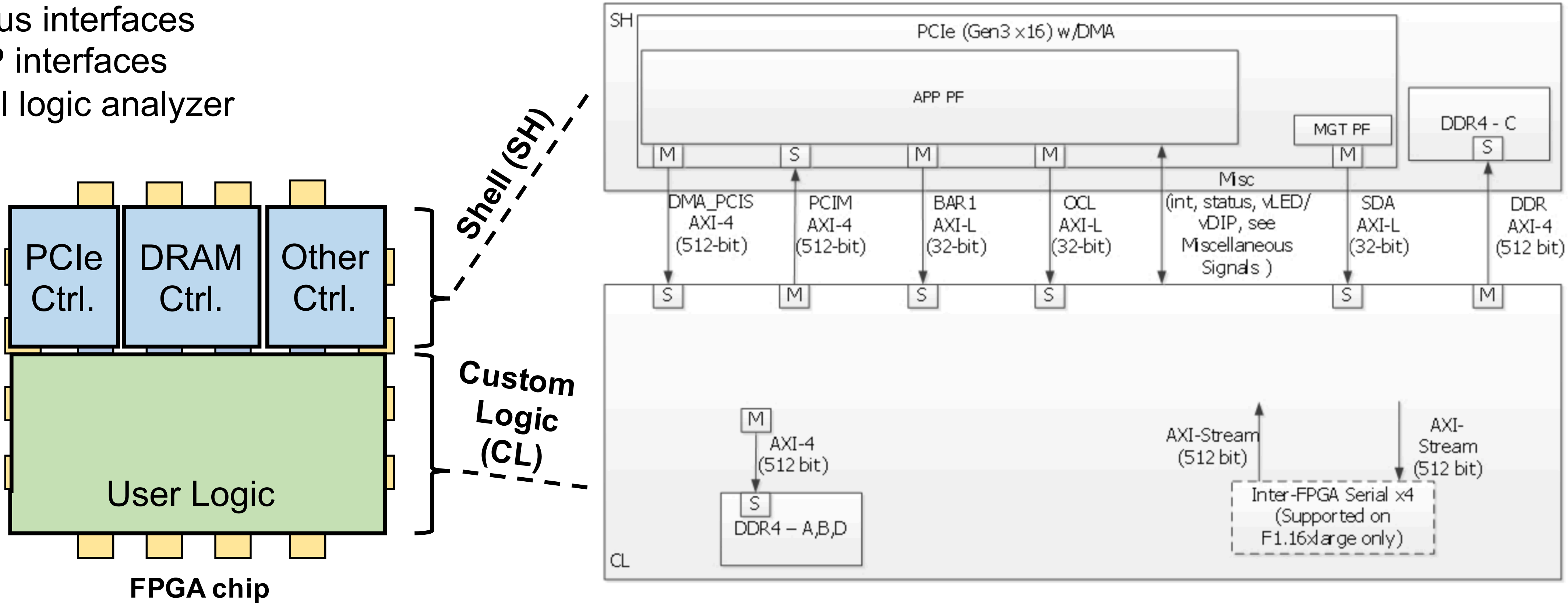
Then locally or remotely finish the development:

- Use HDK locally (if have license) to make the design into custom logic (CL) and finish testing and synthesis of whole design
- Start FPGA Developer AMI with the HDK and make the design into CL and finish testing and synthesis

Custom Logic (CL) is basically user's hardware design connected to AXI ports that go to PCIe and possibly DRAM modules – developing user's module as AXI module is almost required

FPGA Shell Interface and User's Custom Logic

- Using Cloud FPGAs requires some standard modules
  - PCIe controller to communicate with the server
  - DRAM controller to use DRAM modules
  - AXI bus interfaces
  - QSFP interfaces
  - Virtual logic analyzer
  - …

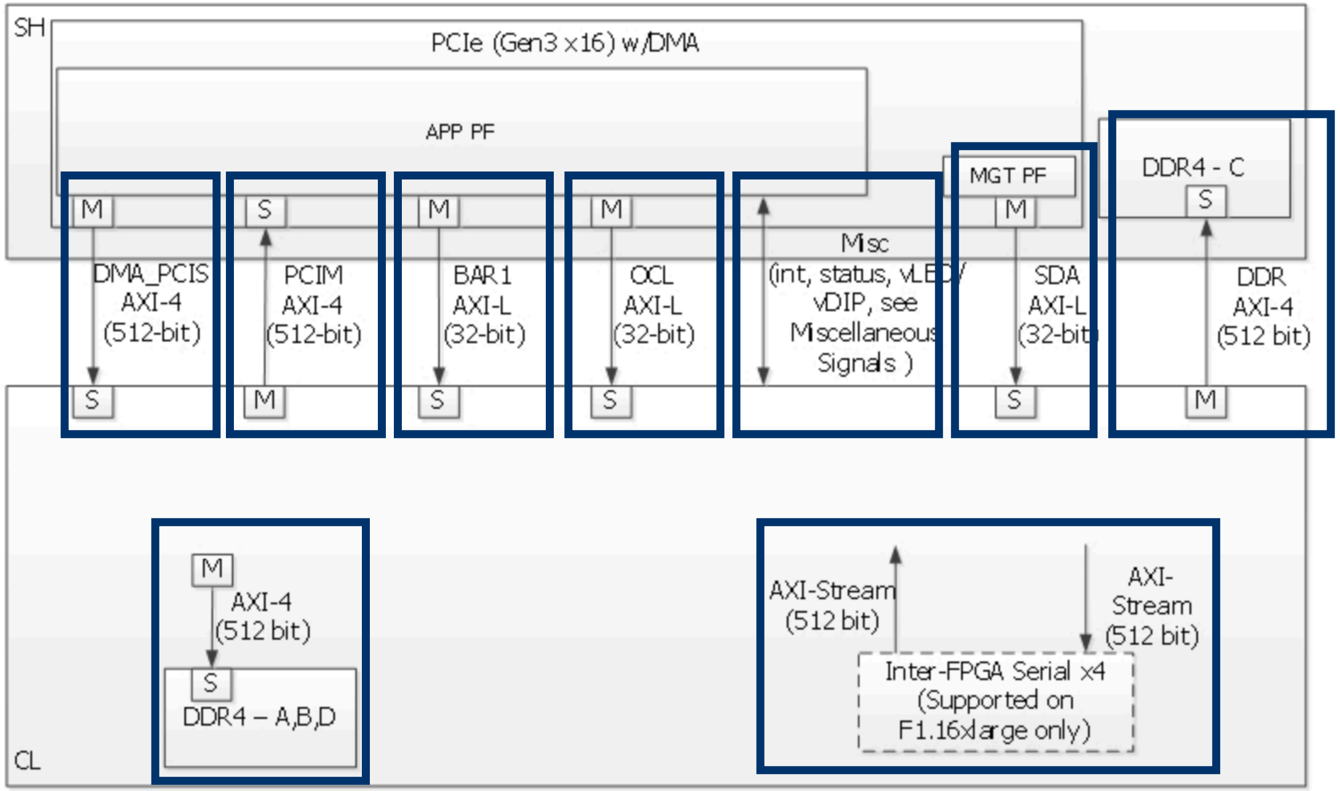


Block diagram from [2]

FPGA Shell Interface and User's Custom Logic

Most of the communication between Shell and the Custom Logic is done through AXI bus

- Different variants of AXI are used
  - AXI4 512-bit
  - AXI4-Lite 32-bit
  - AXI4-Stream 512-bit
- Some other signals are just wires coming into the CL, or register values going out of the CL
- Advantage is standard AXI interface
- Disadvantage is the fixed bit width
  - Hardware may generate data much faster than can be moved off chip



Block diagram from [2]

Share:
bit.ly/cloudfpga

FPGA Shell Address Map

FPGAs are attached by PCIe to the servers

- Each FPGA "Slot" presents a single FPGA with two PCIe Physical Functions (PFs)
- Each PF has multiple Base Address Registers (BARs)

F1 servers have up to 8 slots, e.g. 2x instance has only 1 FPGA which is on slot 0, 16x instance has 8 FPGAs on slots 0 to 7

PCIe (and PCI) devices are identified by BDF (Bus:Device.Function) notation, e.g. 00:02.0

- **Physical BDF of real devices**
- **Virtual BDF exposed to the VM**

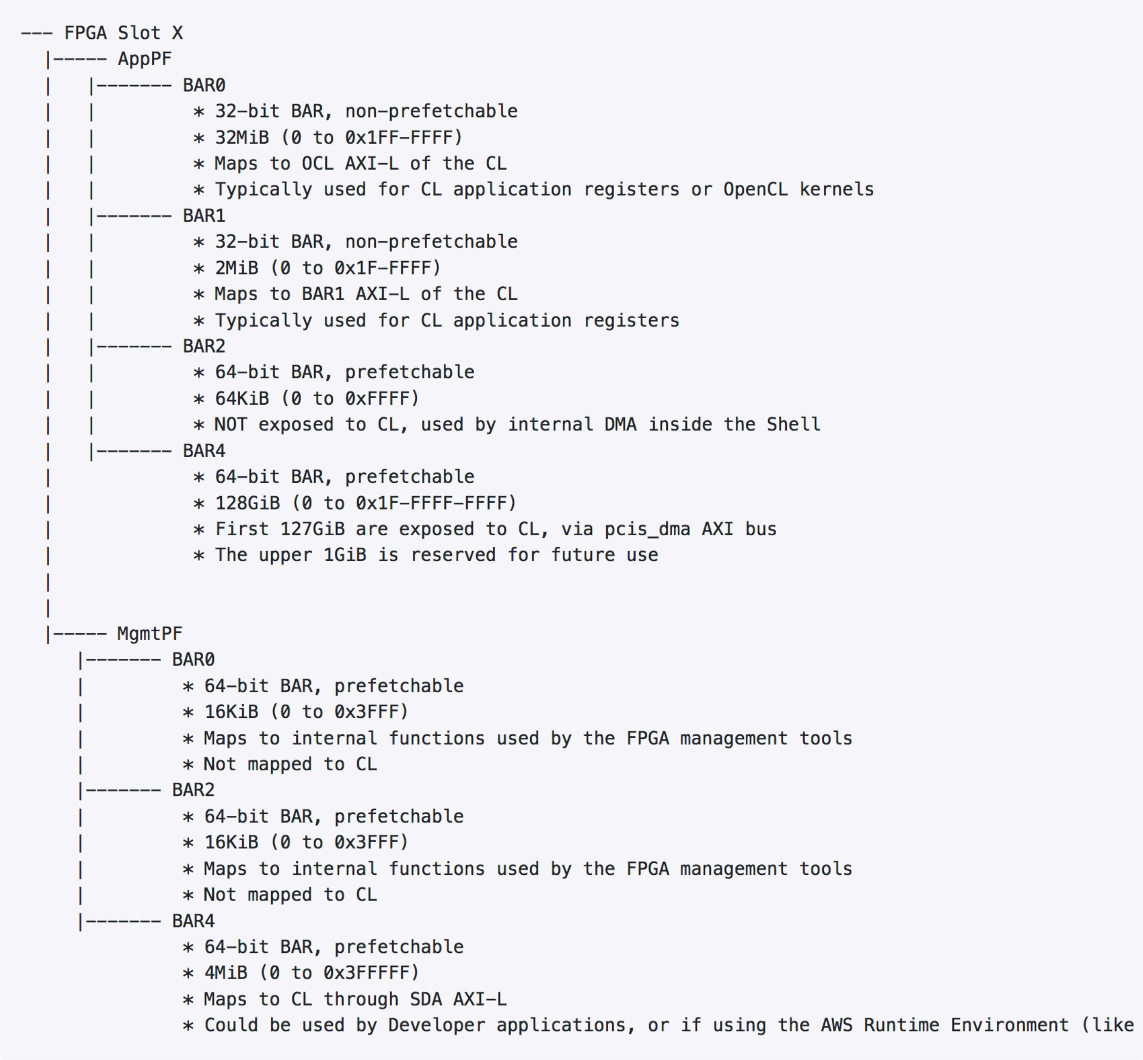
- The BARs are mapped to the instance's memory-mapped I/O (MMIO) space
  - Writing to the specific address range will cause data to be sent to PCIe, not memory
  - Reading from a specific address range will cause data to come from PCIe, not memory
- Addresses need to be mapped to the Linux kernel or a user-space application before accessing them
  - Kernel mapping for DMA related BARs
  - User-space mapping for some management and simple **`peek()`** and **`poke()`** communication

FPGA Shell Address Map

- Each FPGA slot is associated with two PCIe physical functions
- Each has multiple BARs

- BARs have specified sizes, but their actual address is fixed when the VM is started

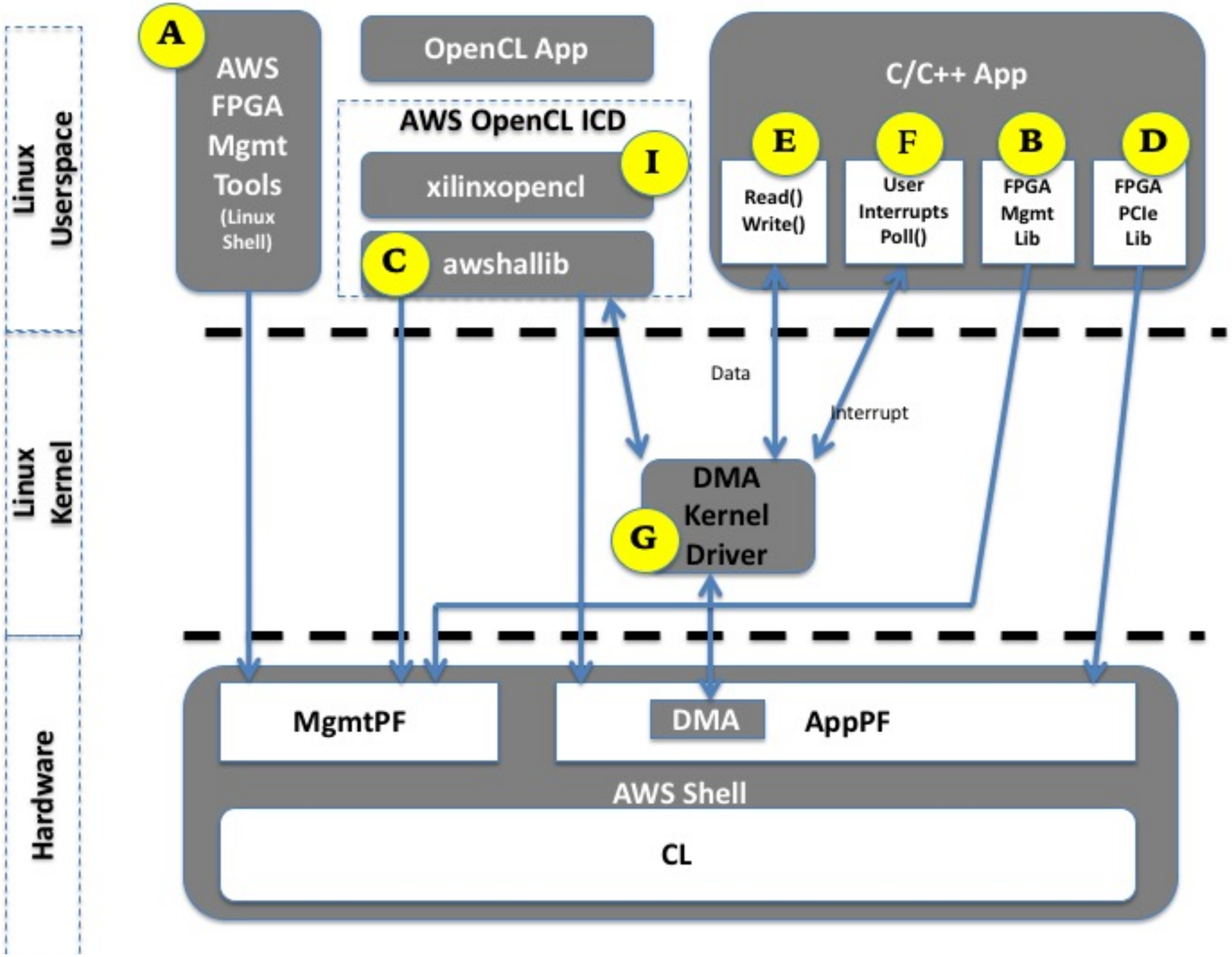
- Not all addresses in BARs are used if the FPGA is not configured with the corresponding functionality



```
--- FPGA Slot X
  |----- AppPF
  |   |------- BAR0
  |   |         * 32-bit BAR, non-prefetchable
  |   |         * 32MiB (0 to 0x1FF-FFFF)
  |   |         * Maps to OCL AXI-L of the CL
  |   |         * Typically used for CL application registers or OpenCL kernels
  |   |------- BAR1
  |   |         * 32-bit BAR, non-prefetchable
  |   |         * 2MiB (0 to 0x1F-FFFF)
  |   |         * Maps to BAR1 AXI-L of the CL
  |   |         * Typically used for CL application registers
  |   |------- BAR2
  |   |         * 64-bit BAR, prefetchable
  |   |         * 64KiB (0 to 0xFFFF)
  |   |         * NOT exposed to CL, used by internal DMA inside the Shell
  |   |------- BAR4
  |             * 64-bit BAR, prefetchable
  |             * 128GiB (0 to 0x1F-FFFF-FFFF)
  |             * First 127GiB are exposed to CL, via pcis_dma AXI bus
  |             * The upper 1GiB is reserved for future use
  |
  |
  |----- MgmtPF
     |------- BAR0
     |         * 64-bit BAR, prefetchable
     |         * 16KiB (0 to 0x3FFF)
     |         * Maps to internal functions used by the FPGA management tools
     |         * Not mapped to CL
     |------- BAR2
     |         * 64-bit BAR, prefetchable
     |         * 16KiB (0 to 0x3FFF)
     |         * Maps to internal functions used by the FPGA management tools
     |         * Not mapped to CL
     |------- BAR4
               * 64-bit BAR, prefetchable
               * 4MiB (0 to 0x3FFFFF)
               * Maps to CL through SDA AXI-L
               * Could be used by Developer applications, or if using the AWS Runtime Environment (like
```

Address map from [3]

Programmer's View of the Custom Logic



Software has multiple ways to communicate with the hardware running on the FPGA:

A. Command line tools
B. Management library (C or Python)
C. OpenCL related
D. PCIe library (C or Python)
E. DMA interface – requires kernel driver
F. Interrupts – requires kernel driver
G. Kernel DMA driver
  - XDMA kernel driver
  - XOCL kernel driver, OpenCL related
I. OpenCL related

Image from [4]

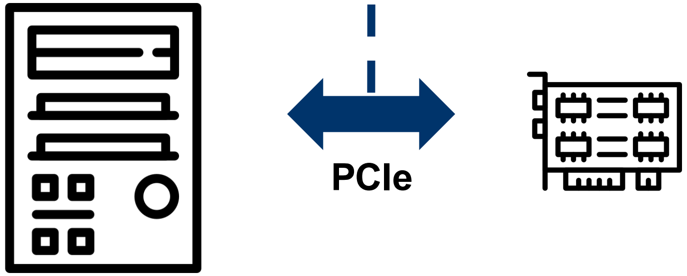
Program Interaction with Hardware on FPGA

- Read or write 32-bit values from registers in the CL (PCIe lib.)
- “Burst” read or write 32-bit values form registers in the CL (PCIe lib.)
- DMA data between server’s DRAM and the FPGA board (DMA lib.)

- No explicit interaction, only setup
- No explicit interaction, only setup

- CL needs to have registers and AXI state machine to respond to reads or writes
- CL needs registers and state machine to handle accesses to contiguous addresses
- CL needs state machine and registers or use DRAM for DMA data transfers

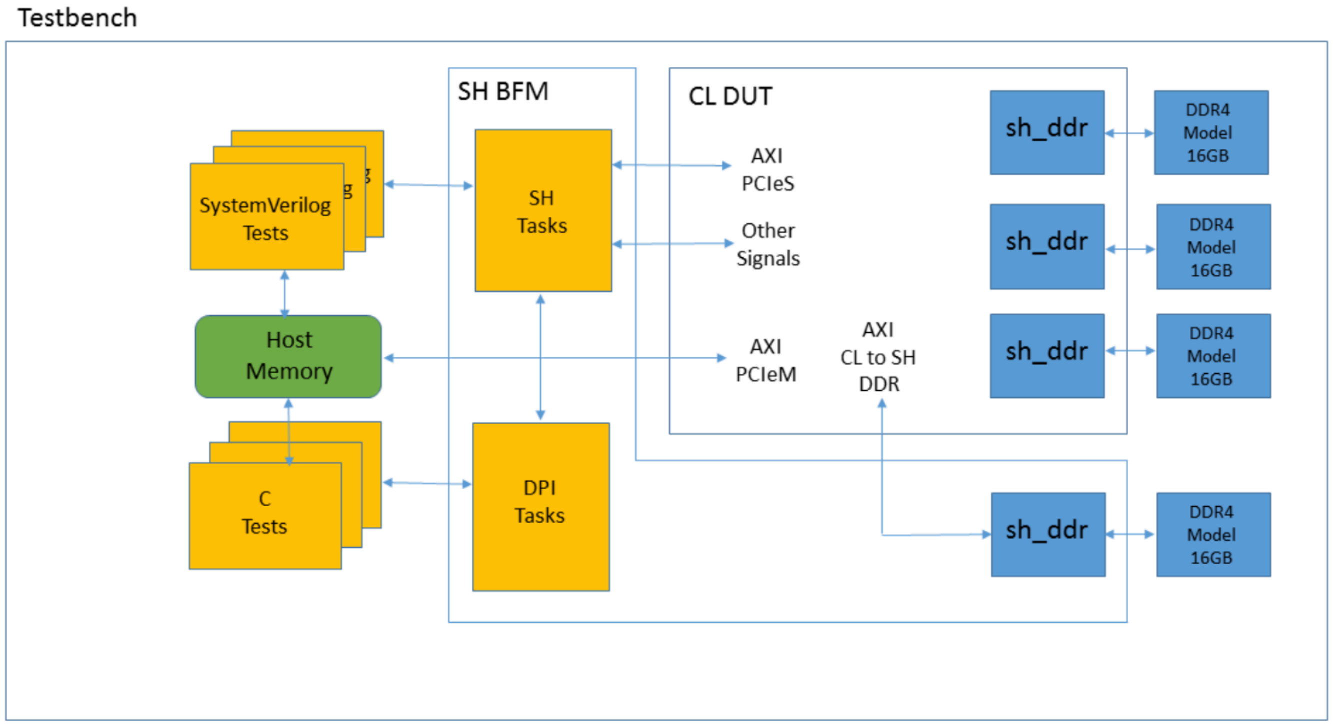
- Optional CL logic for initiating DMA transfers
- Optional CL logic for FPGA-to-FPGA communication (future)



Simulating Custom Logic

Like any hardware design, it needs to be simulated to check functionality, find bugs, etc.

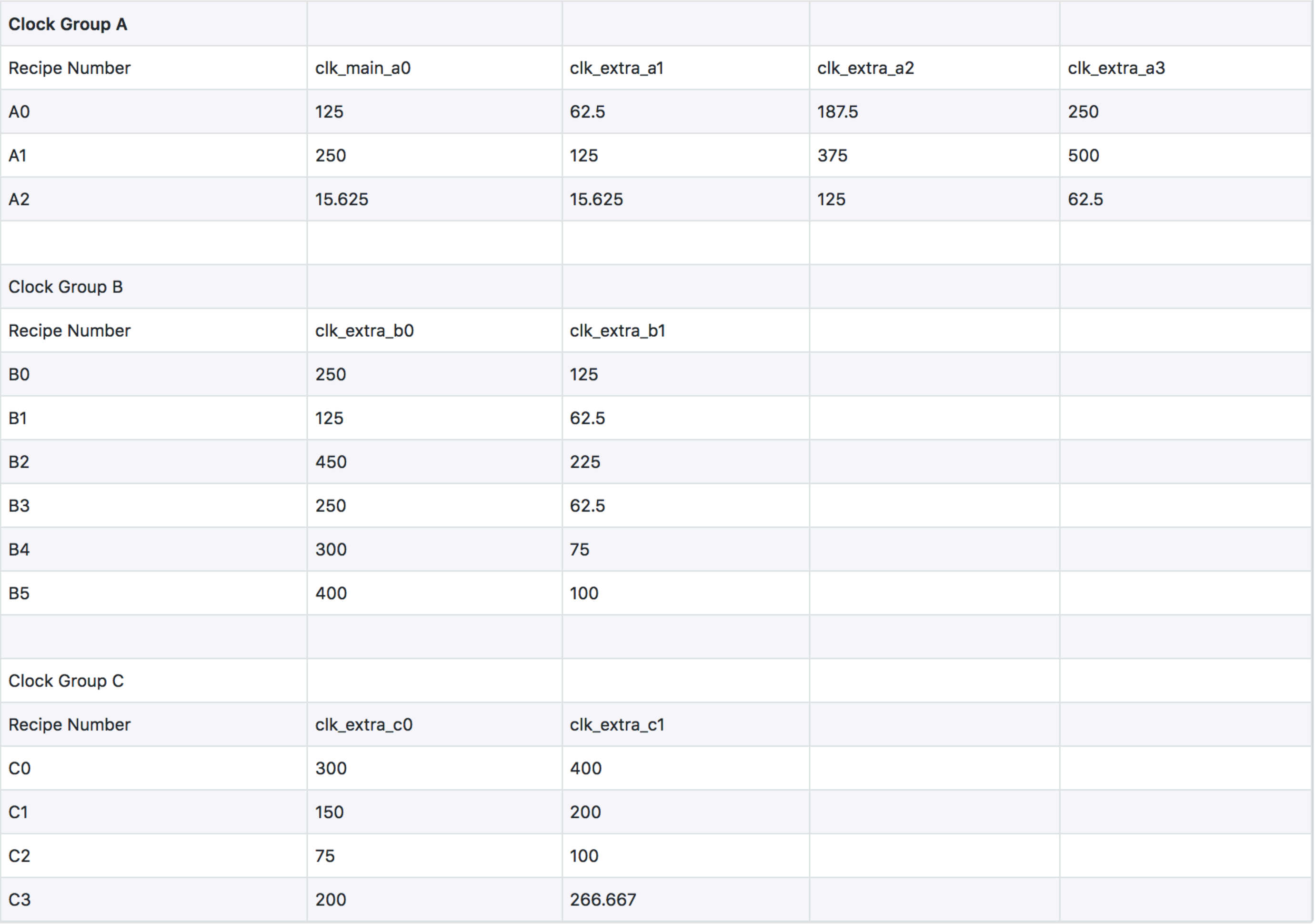
- Amazon provides module to generate the testbenches
  - Include custom hardware (user's CL)
  - Include corresponding software
- Less expensive and saves time to run simulation rather than make AFIs and test on FPGAs
- Testbench simulates how the PCIe and other components to generate AXI signals based on software's operation
- Also simulate DRAM operation
- Any AXI replies, virtual LED updates, etc., are sent back to the software running in simulation



Timing and Available Custom Logic Clocks

Once timing of the CL is found, appropriate clock needs to be used

- Shell (set via configuration file) outputs a number of clocks
- Default is 250Mhz
- Clocks available up to 500Mhz



Clock Group A				
Recipe Number	clk_main_a0	clk_extra_a1	clk_extra_a2	clk_extra_a3
A0	125	62.5	187.5	250
A1	250	125	375	500
A2	15.625	15.625	125	62.5
Clock Group B				
Recipe Number	clk_extra_b0	clk_extra_b1		
B0	250	125		
B1	125	62.5		
B2	450	225		
B3	250	62.5		
B4	300	75		
B5	400	100		
Clock Group C				
Recipe Number	clk_extra_c0	clk_extra_c1		
C0	300	400		
C1	150	200		
C2	75	100		
C3	200	266.667		

Runtime Debugging: Virtual LEDs and DIP Switches

- There are virtual LEDs and DIP switches that can be used to control and monitor users' CL design
- There are 16 virtual LEDs and 16 virtual DIP switches

Physical LEDs and DIP switches on Altera board

- Virtual LEDs are connected to 16 output wires going to user's CL
  - Can be driven from the CL logic to the SH from **`cl_sh_status_vled[15:0]`** signal
- Virtual DIP switches are connected to 16 input registers going to user's CL
  - Are driven from the SH to the CL logic to **`sh_cl_status_vdip[15:0]`** signal

With cloud FPGAs there is no physical access to FPGAs, so virtual versions of LEDs, switches, etc. are needed

Users can an use the command line commands **`fpga-get-virtual-led`** to read the virtual LED values, and **`fpga-set-virtual-dip-switch`** to set the virtual DIP switch values on the Shell-to-CL interface.

Altera DE1 mage from https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/terasic-inc-/board/altera-de1-board.html

Runtime Debugging: Virtual JTAG

JTAG (named after Joint Test Action Group) is an industry standard for verifying electronic designs, typically embedded systems or system-on-a-chip after manufacturing

- A simple serial interface used for programming a devices, or reading their state
- Common application is to use JTAG to write some data or “program” a device
    - Most FPGAs are programmed by JTAG via USB-to-JTAG cable
- Multiple devices can be connected to same JTAG port
    - E.g. there is 1 JTAG port for a system-on-a-chip giving access to all the components

Example JTAG Scan Chain

- JTAG pins
    - *TMS* (Test Mode Select)
    - *TCK* (Test Clock)
    - *TDI* (Test Data In)
    - *TDO* (Test Data Out)
- Use device ID to select device, common commands can be read, write, change function

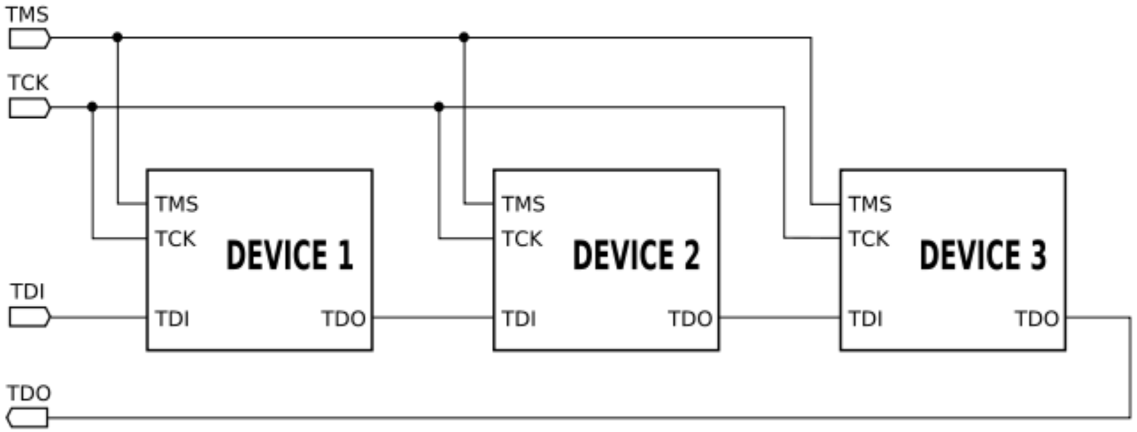
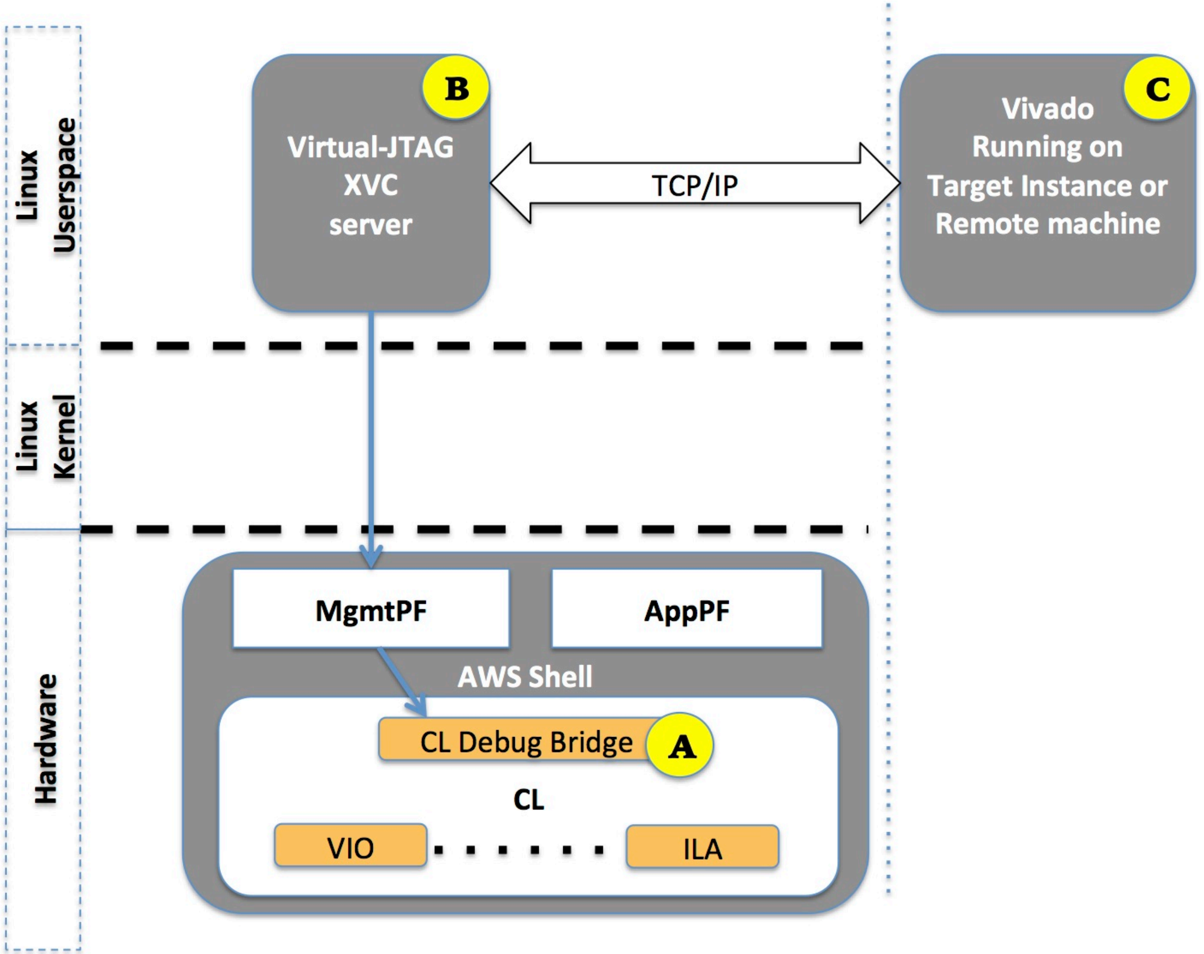


Image from https://en.wikipedia.org/wiki/JTAG

Runtime Debugging: Virtual JTAG



Virtual JTAG creates JTAG interface into the logic inside the CL

- Separate from JTAG used to program the FPGA chip

Use with Vivado modules for runtime debugging

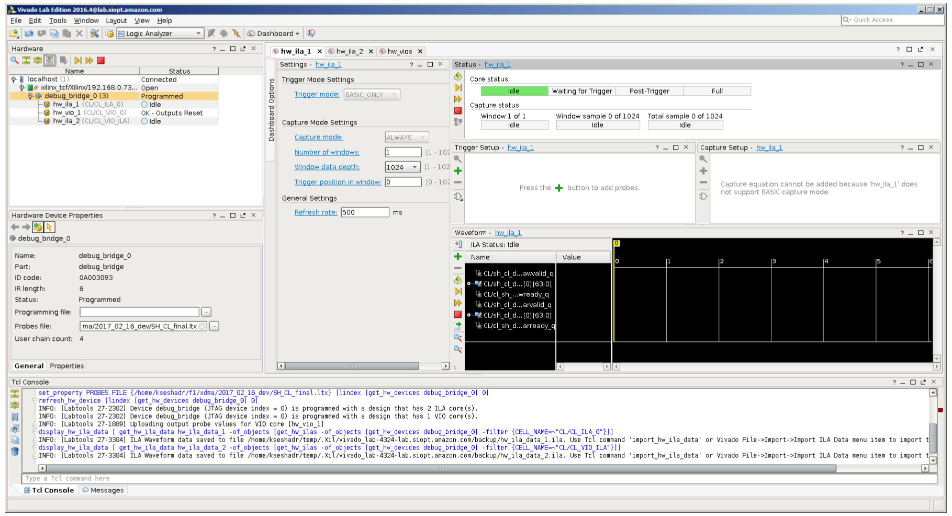
A. CL Debug Bridge receives JTAG commands
B. Virtual JTAG server receives remote JTAG commands and passes them to the FPGA
C. JTAG commands are sent, for example, by Vivado software

Image from [4]

Runtime Debugging: Virtual JTAG

- Integrated Logic Analyzer (ILA) is a IP core that behaves like a logic analyzer, can capture real-time changes in values of different signals and send them for debugging purpose to a waveform viewer
- Virtual Input/Output (VIO) is an IP core that behaves like a input or output pin, but instead of data coming or going to physical FPGA pin, it goes to virtual pin that can be written or read by JTAG commands

Sample image of Vivado with an ILA output:



Name	Status
localhost (1)	Connected
xilinx_tcf/Xilinx/192.168.0.73...	Open
debug_bridge_0 (3)	Programmed
hw_ila_1 (CL/CL_ILA_0)	○ Idle
hw_vio_1 (CL/CL_VIO_0)	OK - Outputs Reset
hw_ila_2 (CL/CL_VIO_ILA)	○ Idle

Hardware Device Properties

debug_bridge_0

Name: debug_bridge_0
Part: debug_bridge
ID code: 0A003093
IR length: 6
Status: Programmed
Programming file:
Probes file: ma/2017_02_16_dev/SH_CL_final.ltx
User chain count: 4

Settings - hw_ila_1

Trigger Mode Settings

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode: ALWAYS
Number of windows: 1 [1 - 102
Window data depth: 1024 [1 - 102
Trigger position in window: 0 [0 - 102

General Settings

Refresh rate: 500 ms

Status - hw_ila_1

Core status: Idle | Waiting for Trigger | Post-Trigger | Full

Capture status
Window 1 of 1: Idle
Window sample 0 of 1024: Idle
Total sample 0 of 1024: Idle

Trigger Setup - hw_ila_1

Press the ✚ button to add probes.

Capture Setup - hw_ila_1

Capture equation cannot be added because 'hw_ila_1' does not support BASIC capture mode.

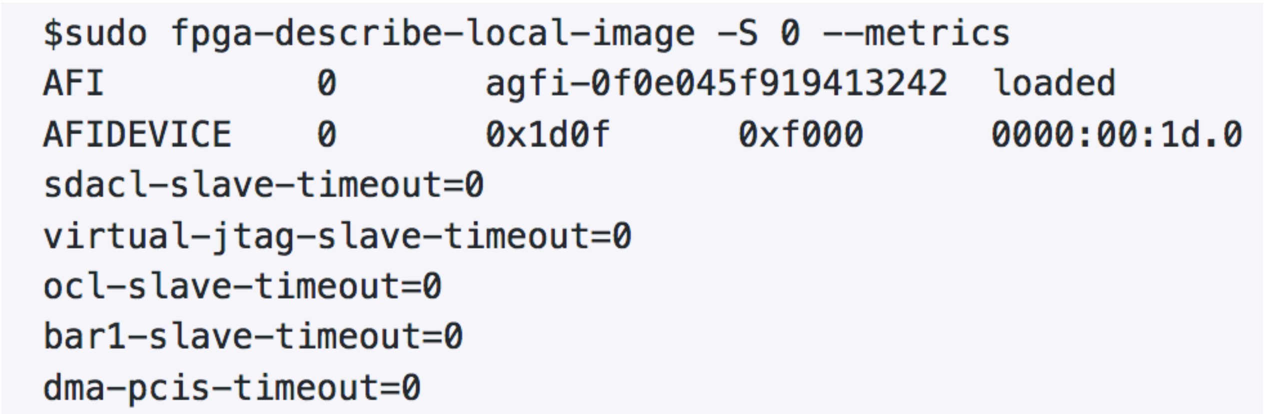
```
set_property PROBES.FILE {/home/kseshadr/f1/xdma/2017_02_16_dev/SH_CL_final.ltx} [lindex [get_hw_devices debug_bridge_0] 0]
refresh_hw_device [lindex [get_hw_devices debug_bridge_0] 0]
INFO: [Labtools 27-2302] Device debug_bridge (JTAG device index = 0) is programmed with a design that has 2 ILA core(s).
INFO: [Labtools 27-2302] Device debug_bridge (JTAG device index = 0) is programmed with a design that has 1 VIO core(s).
INFO: [Labtools 27-1889] Uploading output probe values for VIO core [hw_vio_1]
display_hw_ila_data [ get_hw_ila_data hw_ila_data_1 -of_objects [get_hw_ilas -of_objects [get_hw_devices debug_bridge_0] -filter {CELL_NAME=~"CL/CL_ILA_0"}]]
INFO: [Labtools 27-3304] ILA Waveform data saved to file /home/kseshadr/temp/.Xil/vivado_lab-4324-lab.siopt.amazon.com/backup/hw_ila_data_1.ila. Use Tcl command 'import_hw_ila_data' or Vivado File->Import->Import ILA Data menu item to import t
display_hw_ila_data [ get_hw_ila_data hw_ila_data_2 -of_objects [get_hw_ilas -of_objects [get_hw_devices debug_bridge_0] -filter {CELL_NAME=~"CL/CL_VIO_ILA"}]]
INFO: [Labtools 27-3304] ILA Waveform data saved to file /home/kseshadr/temp/.Xil/vivado_lab-4324-lab.siopt.amazon.com/backup/hw_ila_data_2.ila. Use Tcl command 'import_hw_ila_data' or Vivado File->Import->Import ILA Data menu item to import t
Type a Tcl command here
```

Image from
https://en.wikipedia.org/wiki/JTAG

Runtime Timeout Issues

The shell provides timeout mechanism in case the FPGA design is not responding in time

- Each command ends up being a request on the corresponding AXI bus
  - CL register related AXI bus
  - DMA related AXI bus
- AXI transactions are terminated after 8 us
- According to Amazon's documentation, timeouts can occur for three reasons:
  - The CL doesn't respond to the address (reserved address space)
  - The CL has a protocol violation on AXI which hangs the bus
  - The CL design's latency is exceeding the timeout value
- Command line tools give information about timeouts
  - But will notice in software quickly if, e.g., **`peek()`** and **`poke()`** commands don't work



```
$sudo fpga-describe-local-image -S 0 --metrics
AFI          0       agfi-0f0e045f919413242  loaded
AFIDEVICE    0       0x1d0f      0xf000      0000:00:1d.0
sdacl-slave-timeout=0
virtual-jtag-slave-timeout=0
ocl-slave-timeout=0
bar1-slave-timeout=0
dma-pcis-timeout=0
```

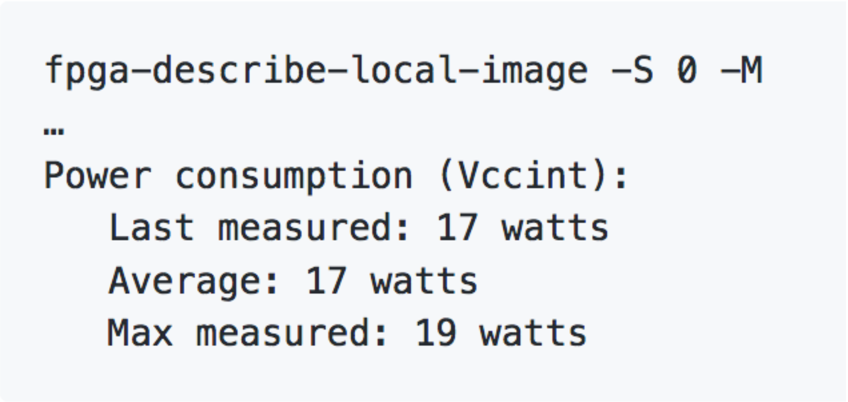
Runtime Power Analysis and Protection Features

The servers with FPGAs need to be protected from the FPGAs using too much power, which could cause shutdown or physical damage to FPGAs and/or the server

- `afi-power-warning` will be generated if power is above 85 Watts
  - Xilinx Virtex UltraScale+ FPGA VCU1525 and Xilinx Alveo U200/U250/U280 Accelerator Cards are rated with Thermal Design Power (TDP) of 225 Watts

- `afi-power-violation` will be generated if certain power threshold is breached
  - Value not clearly stated by Amazon
  - Likely depends on current server load
  - All clocks will be throttled or disabled on power violation
  - Unclear if design is unloaded or FPGA reset if clock throttling or stopping clocks does not help

- Command line tools can show average power usage, updated ever 1 min. after that image is loaded



```
fpga-describe-local-image -S 0 -M
…
Power consumption (Vccint):
   Last measured: 17 watts
   Average: 17 watts
   Max measured: 19 watts
```

Design Time Evaluation of Power Usage

Dynamic power of a circuit can be approximated by $\mathbf{P} = CV^2 f$ where C is capacitance, f is frequency, and V is voltage

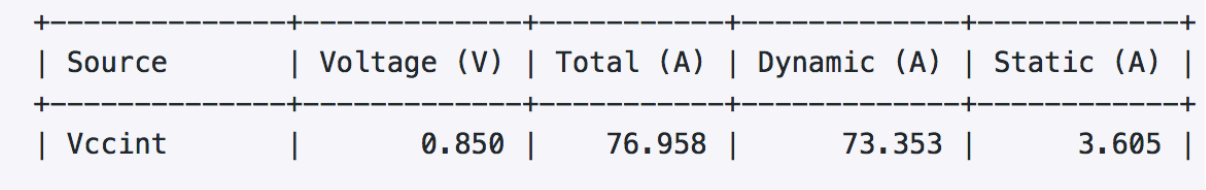
As FPGA chips shrink below 10nm, static power becomes an issue as well

- *C* – reduce capacitance by having smaller or simpler circuit
- *V* – voltage is fixed by the FPGA chip
- *f* – frequency depends on clock used (usually want fastest clock, but can select slower clock to save on dynamic power)

Design and architectural options:

- Use multiple clock domains, some parts of design run slower
- Use flip-flops with enable and only enable when needed
- Use clock gating, turn off clock to parts of design

FPGA tools like Vivado give estimated power for whole design and submodules



```
+--------------+-------------+-----------+-------------+------------+
| Source       | Voltage (V) | Total (A) | Dynamic (A) | Static (A) |
+--------------+-------------+-----------+-------------+------------+
| Vccint       |       0.850 |    76.958 |      73.353 |      3.605 |
```

Amazon F1 SDK

Amazon F1 SDK

The Software Development Kit (SDK) provides tools and libraries that run in the VM and let users interact with the hardware

SDK includes:

- Linux Kernel Drivers
  - XDMA Driver, DMA interface to and from HDK accelerators
  - XOCL Driver, DMA interface with software defined accelerators (HLS designs)
- FPGA Libraries - APIs used by host applications
  - C/C++ library
  - Python bindings
- FPGA Management Tools

`fpga_mgmt`
- Get FPGA status, load image, clear image, etc.
- Read virtual LEDs
- Set virtual dip switches

`fpga_pcie`
- PCIe setup related
- **`peek()`** and **`poke()`** implementations

`fpga_dma` – functions to control Direct Memory Access
- Setup DMA
- Copy data from device
- Copy data to device

SDK information from https://github.com/aws/aws-fpga/#fpgasdk

References

Links to HDK pages from Amazon's AWS git include the version number, some documents seem to be not updated as frequently as others, thus the listed versions are not always the same. Most recent version as of when the slides were made was v1.4.10 for the HDK.

1. "AWS FPGA Hardware Development Kit (HDK), RELEASE V1.4.8" Available at: https://github.com/aws/aws-fpga/blob/master/hdk/README.md
2. "AWS Shell Interface Specification, v1.4.5" Available at: https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md
3. "AWS FPGA PCIe Memory Map, v1.4" Available at: https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Fpga_Pcie_Memory_Map.md
4. "AWS FPGA: Programmer's View of the Custom Logic, v1.4" Available at: https://github.com/aws/aws-fpga/blob/master/hdk/docs/Programmer_View.md
5. "Virtual JTAG for Real-time FPGA Debug, v1.4.4" Available at: https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md
6. "RTL Simulation for Verilog/VHDL Custom Logic Design with AWS HDK, v1.4.10" Available at: https://github.com/aws/aws-fpga/blob/master/hdk/docs/RTL_Simulating_CL_Designs.md