

Jin-Soo Kim  
(jinsoo.kim@snu.ac.kr)

Systems Software &  
Architecture Lab.  
Seoul National University

Fall 2018

# Exceptions and Interrupts



# Exceptional Control Flow

- Conditions under which processor cannot continue normal operation
- Exceptions
  - Generated internally by software executing instructions
    - e.g., INT instruction in x86, Divide-by
  - Synchronous
  - Exception handling is same as interrupt handling
- Interrupts
  - Generated by external hardware devices
    - e.g., Triggered by a signal in INTR or NMI pins (x86)
  - Asynchronous
  - Handler returns to “next” instruction

# Further Classification of Exceptions

## ■ Traps

- Intentional
- System calls, breakpoint traps, special instructions, ...
- Return control to “next” instruction

## ■ Faults

- Unintentional but possibly recoverable
- Page faults (recoverable), protection faults (unrecoverable), ...
- Either re-execute faulting (“current”) instruction or abort

## ■ Aborts

- Unintentional and unrecoverable
- Parity error, machine check, ...

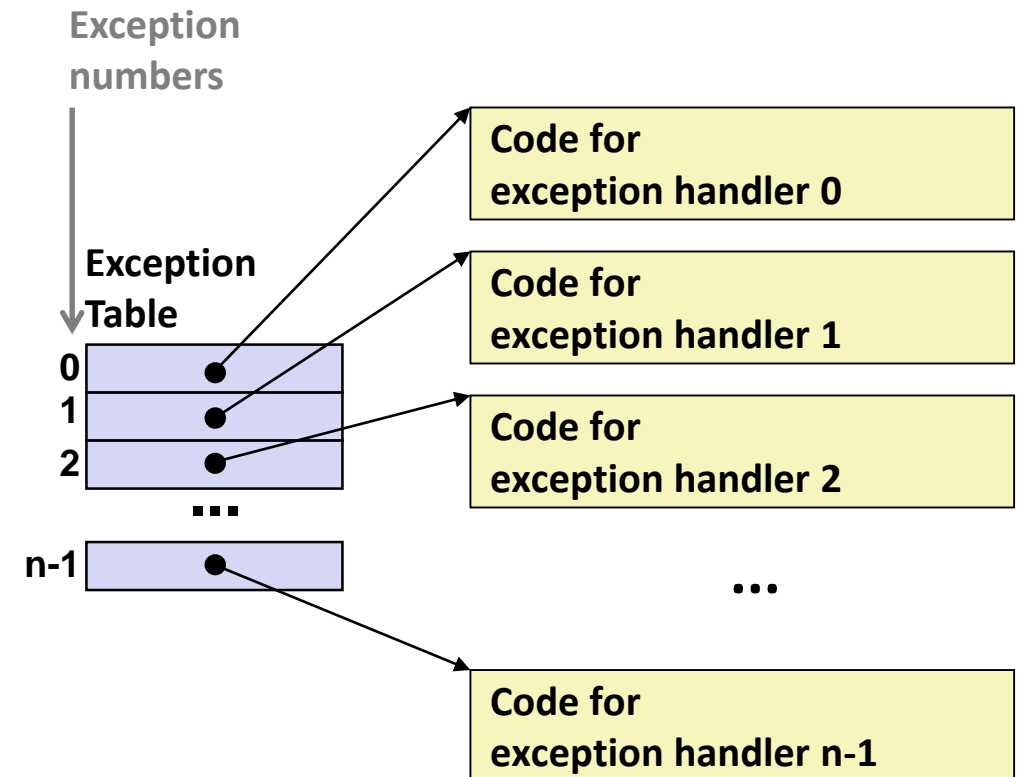
# Handling Exceptions

- Typical desired action

- Complete either current or previous instructions (depends on exception type)
- Discard others
- Call exception handler:  
like an unexpected procedure call

- Exception table

- Each type of event has a unique exception number  $k$ 
  - $k$  = index into exception table (a.k.a. interrupt vector)
- Handler  $k$  is called each time exception  $k$  occurs



# Y86-64 Exceptions

- Detect in Fetch stage

```
jmp  $-1           # Invalid jump target address  
.byte 0xFF         # Invalid instruction code  
halt              # Halt instruction
```

- Detect in Memory stage

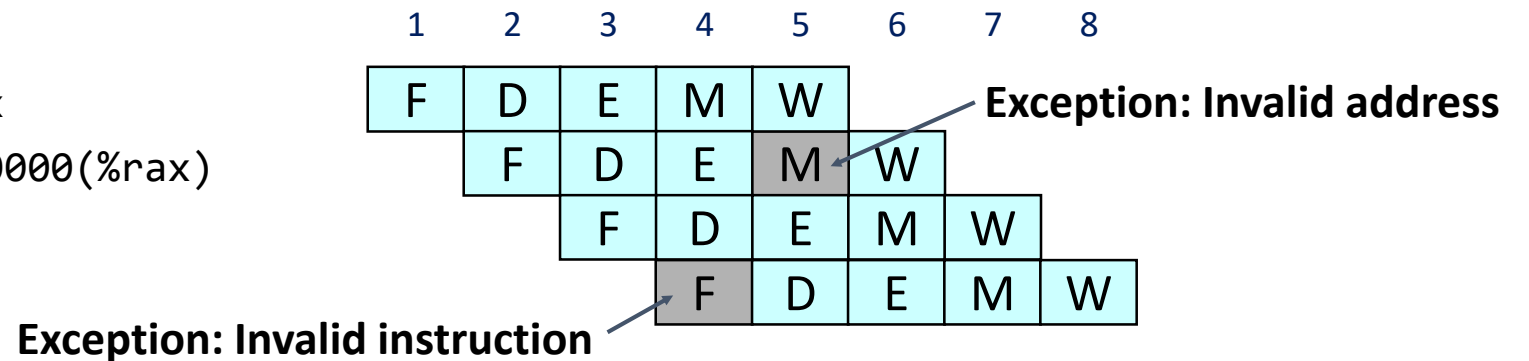
```
rmmovq $rax,0x10000(%rax) # Invalid address  
mrmovq (%rdx), $rax      # Invalid address
```

- Y86-64 implementation: Halt when instruction causes exception

# Exception Handling #1

- Exceptions can be triggered by multiple instructions simultaneously

```
0x000:  irmovq  $100,%rax
0x00a:  rmmovq  %rax,0x10000(%rax)
0x014:  nop
0x015:  .byte 0xFF
```



- **Desired behavior**

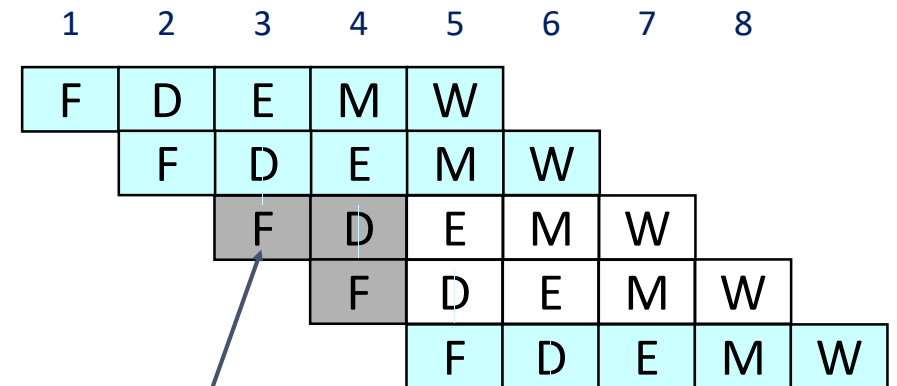
- Just behave like a sequential processor!
- Only the first exception (rmmovq) is reported
  - Put priority on the exception triggered by the instruction that is furthest along the pipeline
- Following instructions should have no effect on processor state

# Exception Handling #2

- Exceptions can be raised in a mispredicted branch path

```
0x000: xorq   %rax, %rax
0x002: jne    t
0x00b: irmovq $1,%rax
0x015: irmovq $2,%rdx
0x01f: halt
0x020: t: .byte 0xFF
```

```
0x000: xorq   %rax,%rax
0x002: jne    t
0x020: t: .byte 0xFF
0x???: (I'm lost!)
0x00b: irmovq $1,%rax
```



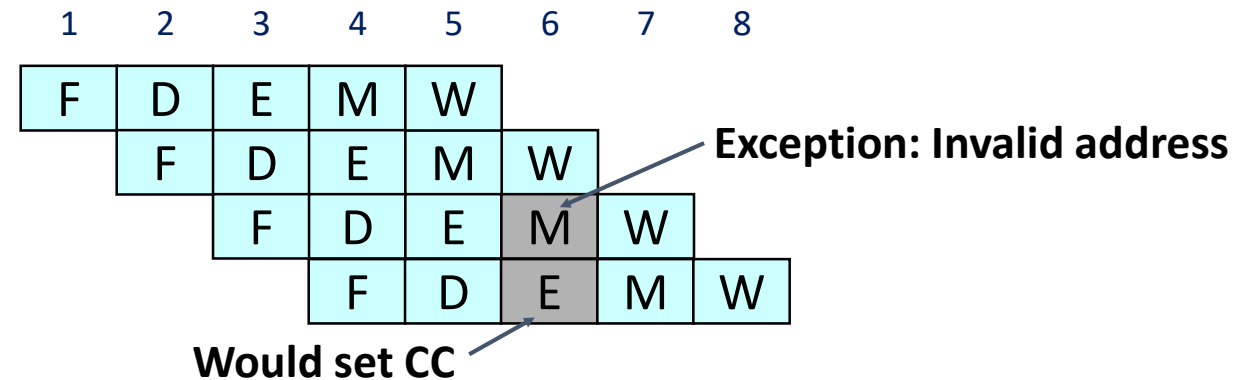
**Exception: Invalid instruction**

- Desired behavior
  - No exception should occur
  - Cancel the excepting instructions

# Exception Handling #3

- System state can be changed by following instructions

```
0x000:  irmovq  $1,%rax
0x00a:  xorq    %rsp,%rsp
0x00c:  pushq   %rax
0x00e:  addq    %rax,%rax
```



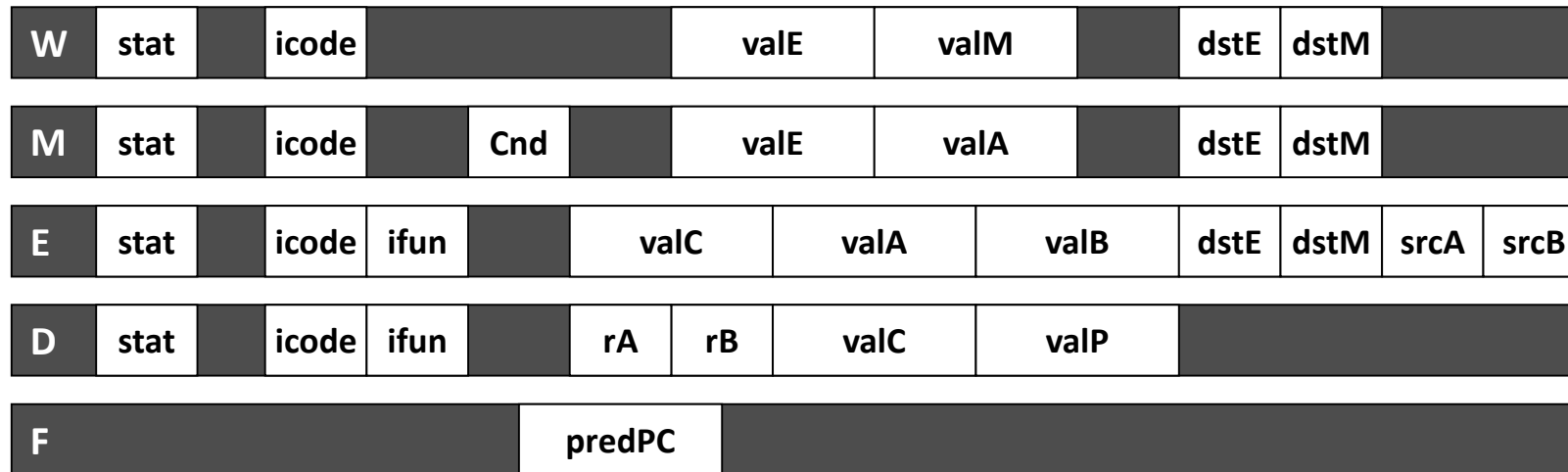
- Desired behavior

- None of the instructions following the excepting instruction should have any effect on the system state
- Update of CC disabled by signals from memory and write back stages (m\_stat and W\_stat)



# Maintaining Exception Ordering

- Add status field to pipeline registers
  - Fetch stage sets to either “AOK”, “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
  - Decode & execute pass values through
  - Memory either passes through or sets to “ADR”
  - Exception triggered only when instruction hits write back



# Exception Handling Logic

- Fetch stage

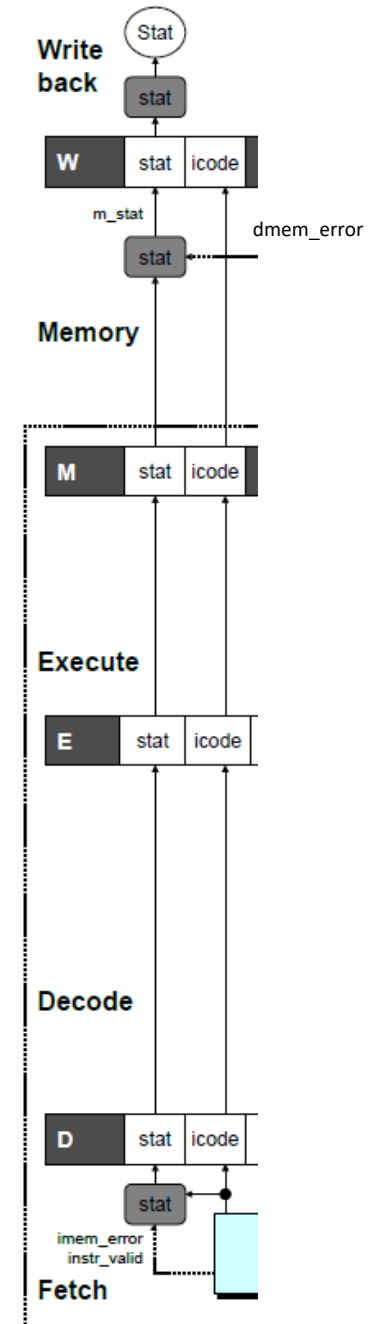
```
# Determine status code for fetched instruction
int f_stat = [
    imem_error : SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];
```

- Memory stage

```
# Update the status
int m_stat = [
    dmem_error : SADDR;
    1 : M_stat;
];
```

- Write back stage

```
# SBUB in earlier stages indicates bubble
int Stat = [
    W_stat == SBUB : SAOK;
    1 : W_stat;
];
```



# Avoiding Side Effects

- Invalid instructions are converted to pipeline bubbles
  - Except stat indicating exception status
- Data memory will not write to invalid address
- Prevent invalid update of condition codes
  - Detect exception in memory stage
  - Disable condition code setting in execute
  - Must happen in same clock cycle
- Handling exception in final stages
  - When detect exception in memory stage: Start injecting bubbles into memory stage on next cycle
  - When detect exception in write back stage: Stall excepting instruction

# Control Logic for State Changes

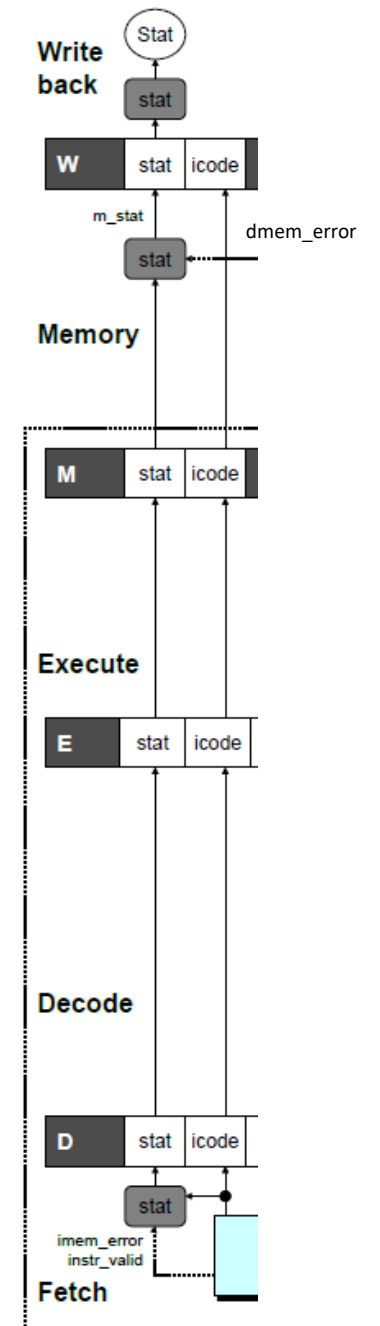
- Setting condition codes

```
# Should the condition codes be updated?  
bool set_cc E_icode == IOPQ &&  
           # State changes only during normal operation  
           !m_stat in { SADR, SINS, SHLT } &&  
           !W_stat in { SADR, SINS, SHLT };
```

- Stage control (also controls updating of memory)

```
# Start injecting bubbles as soon as exception passes through memory stage  
bool M_bubble = m_stat in { SADR, SINS, SHLT } ||  
           W_stat in { SADR, SINS, SHLT };
```

```
# Stall pipeline register W when exception encountered  
bool W_stall = W_stat in { SADR, SINS, SHLT };
```



# The Meltdown Vulnerability

```
# %rcx: kernel address
# %rbx: probe array (256 pages)

xorq %rax, %rax
retry:
movb (%rcx), %al ; raise an exception
shlq $0xc, %rax ; %rax *= 4096
jz retry
movq (%rbx, %rax), %rbx
```

0x0000 0000 0000 0000

User  
(128TB)

0x0000 7fff ffff ffff

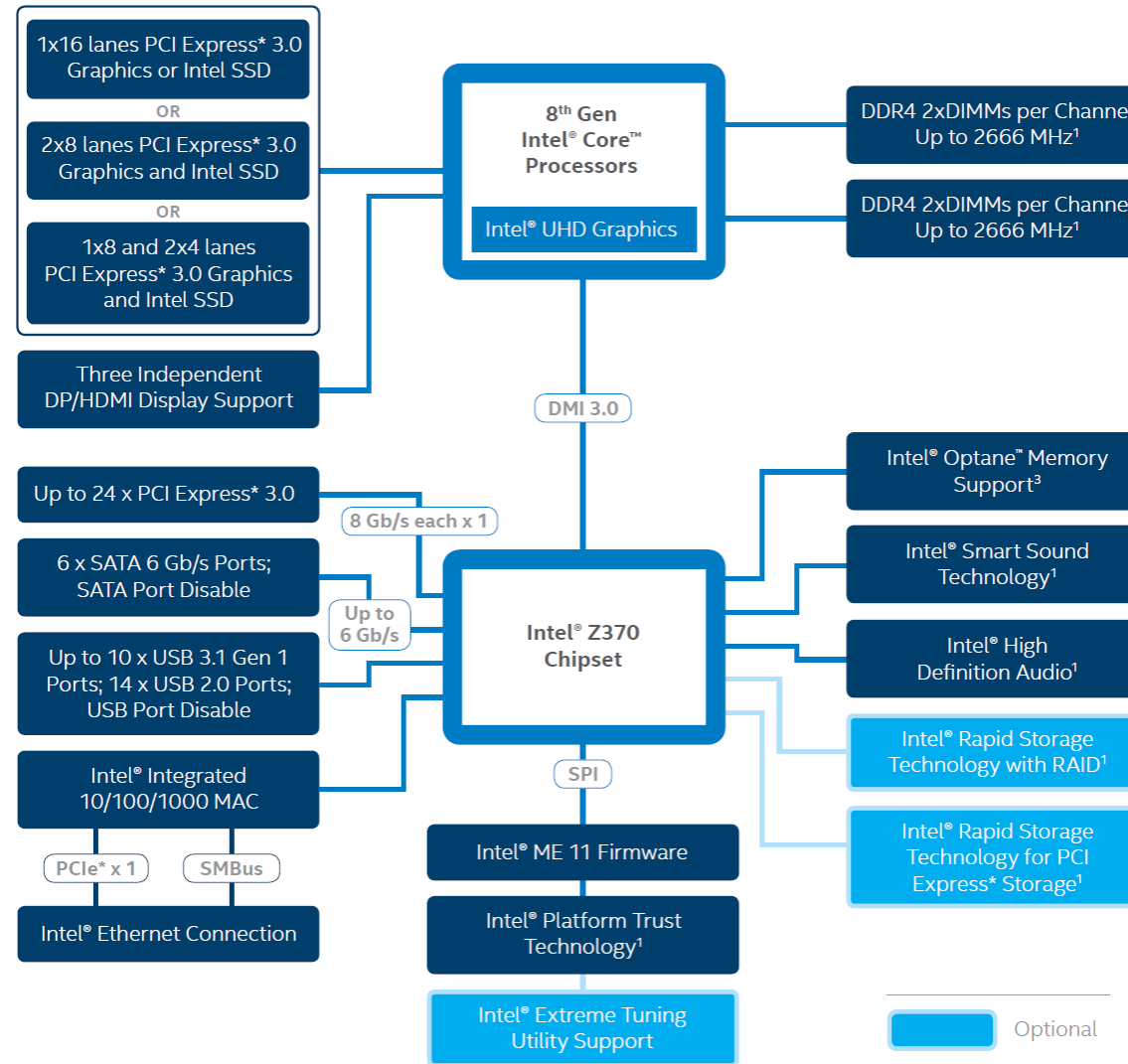
0xffff 8000 0000 0000

Kernel  
(128TB)

0xffff ffff ffff ffff

# Interrupts and DMA

# Modern Computer System

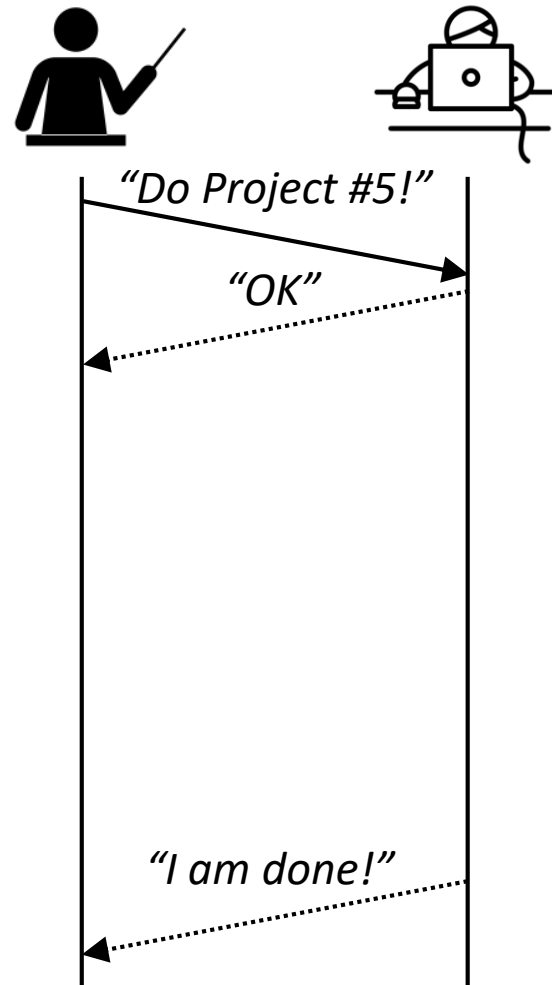
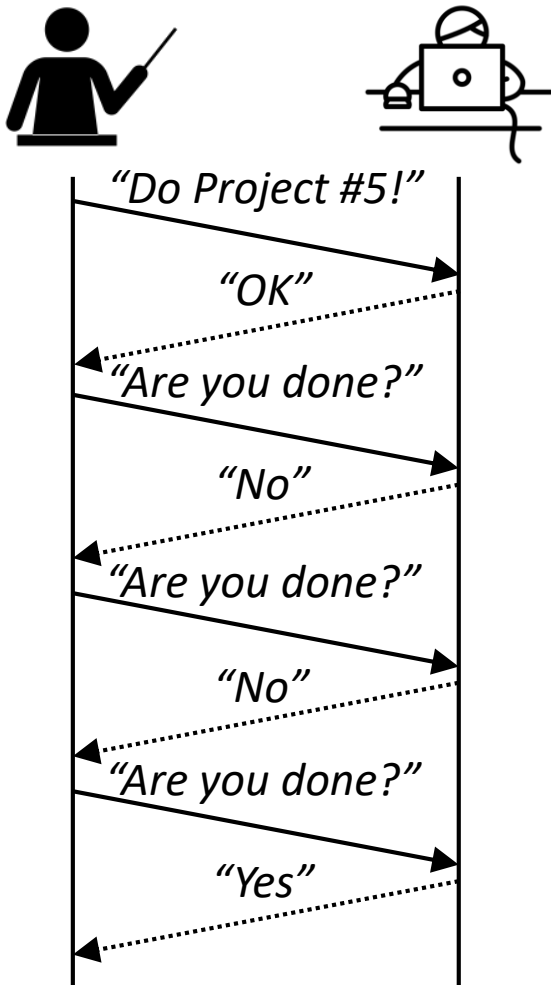


# Handling I/Os

- **How to perform I/Os efficiently?**
  - I/O devices and CPU can execute concurrently
  - Each device controller is in charge of a particular device type
  - Each device controller has a local buffer
  - CPU issues specific commands to I/O devices
  - CPU moves data between main memory and local buffers
- **CPU is a precious resource; it should be freed from time-consuming tasks**
  - Checking whether the issued command has been completed or not
  - Moving data between main memory and device buffers



# Polling vs. Interrupts

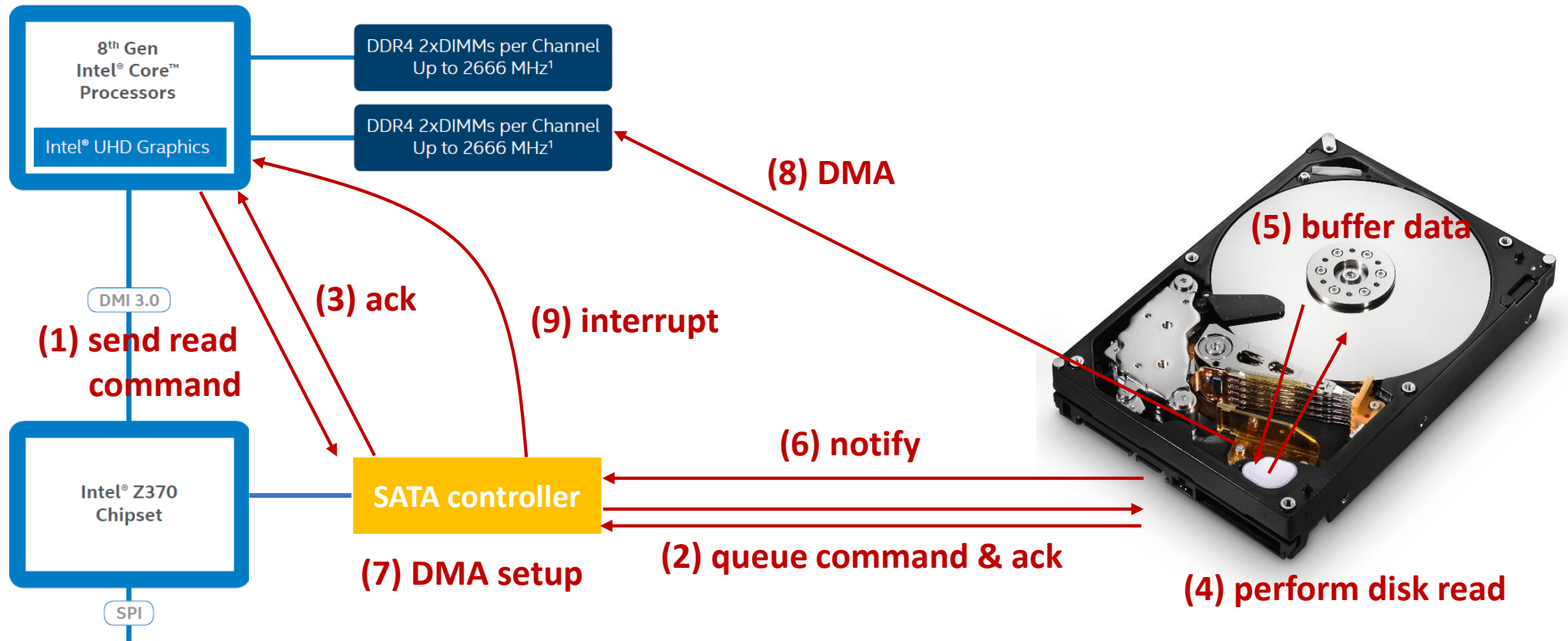


Courtesy: icons by Parkjisun and Cezary Lopacinski from the Noun Project

# Data Transfer Modes

- **Programmed I/O (PIO)**
  - CPU is involved in moving data between I/O devices and memory
  - By special I/O instructions vs. by memory-mapped I/O
- **DMA (Direct Memory Access)**
  - Used for high-speed I/O devices to transmit information at close to memory speeds
  - Device controller transfers blocks of data from the local buffer directly to main memory (or vice versa) without CPU intervention
  - Only an interrupt is generated per block

# Disk I/O Example



# Course Wrap-Up

# What We Have Learned

- Instruction Set Architecture (i.e., abstraction of hardware)
- Representing numbers: integer and floating-point
- x86-64 assembly and how to translate C program into it
- Basic processor organization
- Pipelining
- Branch prediction
- Locality and memory hierarchy
- Caches
- Program optimization for caches
- Virtual memory
- Performance evaluation
- Exceptions and interrupts

# Beyond Pipelining

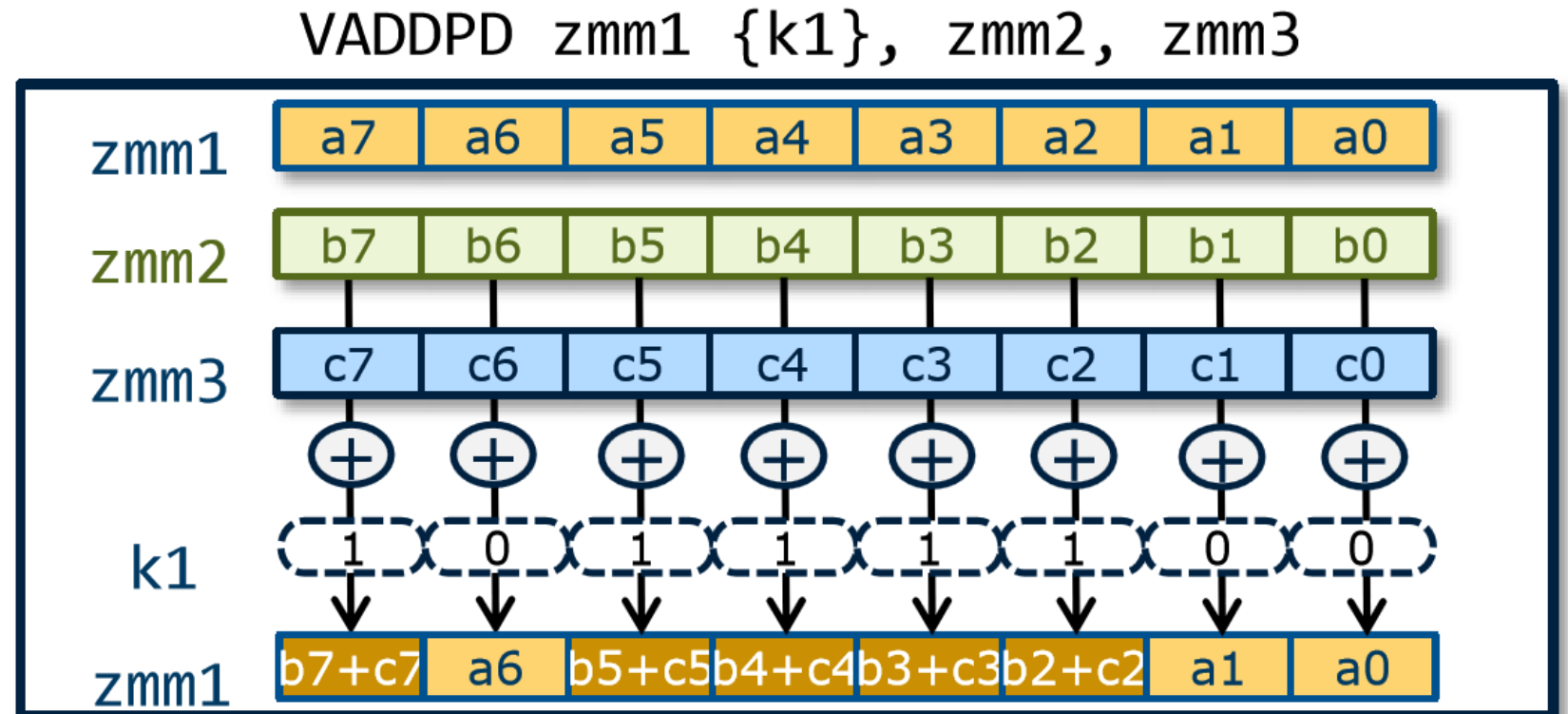
- Hyperthreading
- SIMD (MMX, SSE)
- Vector processor (AVX-512)
  
- Manycore (Xeon Phi: Knights Ferry / Corner / Landing / Hill)
  - Knights Landing (KNL): 72 cores, 4 threads / core
- GPUs
  - Nvidia Titan V (Volta): 5120 CUDA cores + 640 tensor cores
- Accelerators
  - Amazon EC2 F1 instances: up to 8 Xilinx FPGAs (each with 2.5M logic elements)

# Intel AVX-512

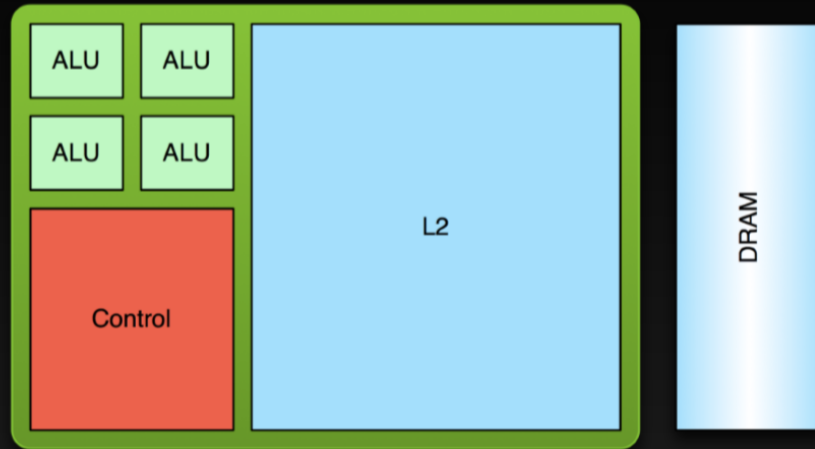
- 32 512-bit registers (%zmm0-31)

- 64 bytes
- 32 words
- 16 doublewords or single-precision FPs
- 8 quadwords or double-precision FPs

- 8 mask registers

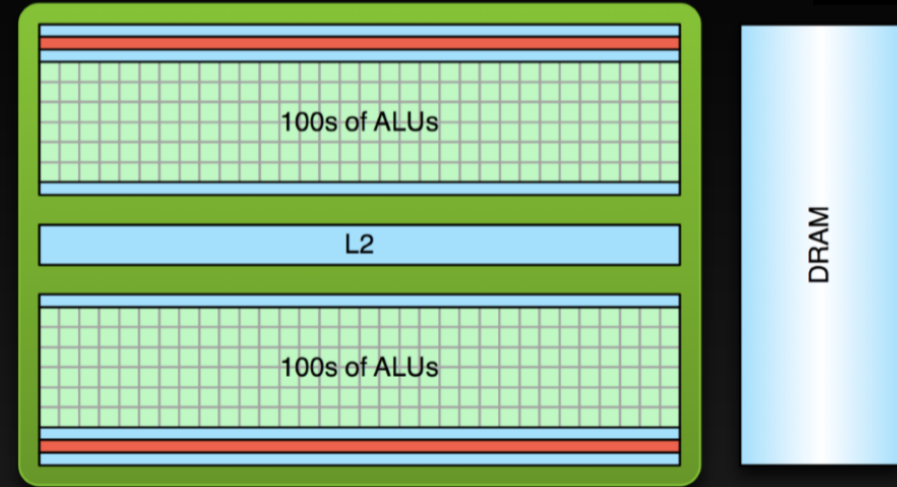


# NVIDIA GPU



## CPU

- **Optimized for low-latency access to cached data sets**
- **Control logic for out-of-order and speculative execution**

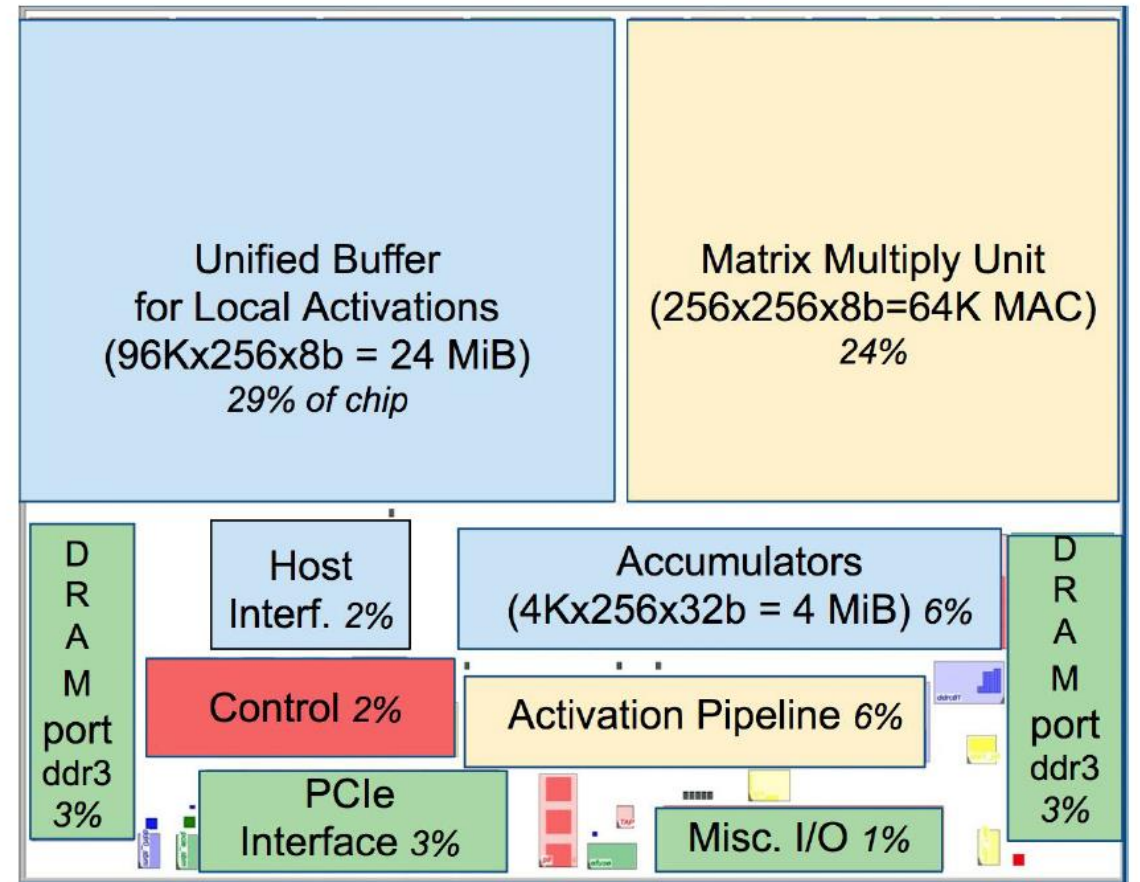
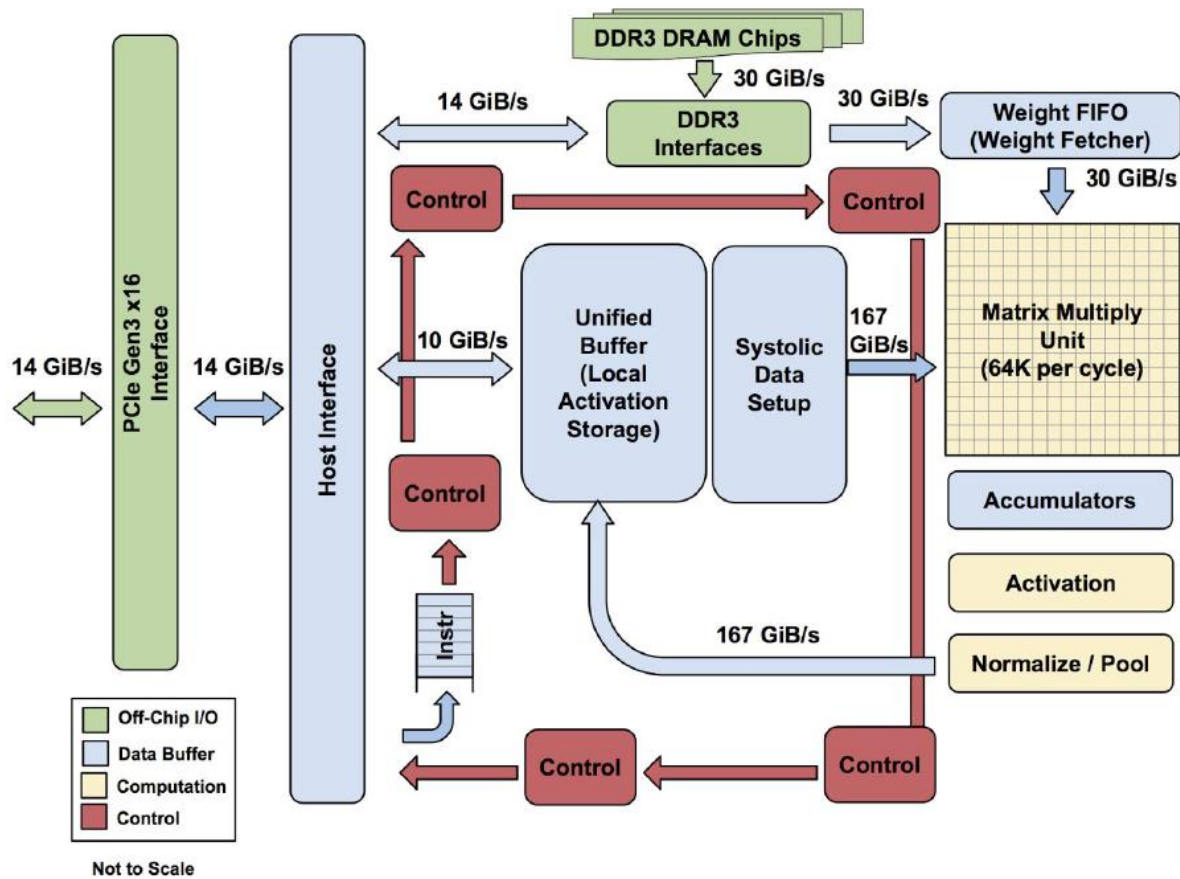


## GPU

- **Optimized for data-parallel, throughput computation**
- **Architecture tolerant of memory latency**
- **More transistors dedicated to computation**



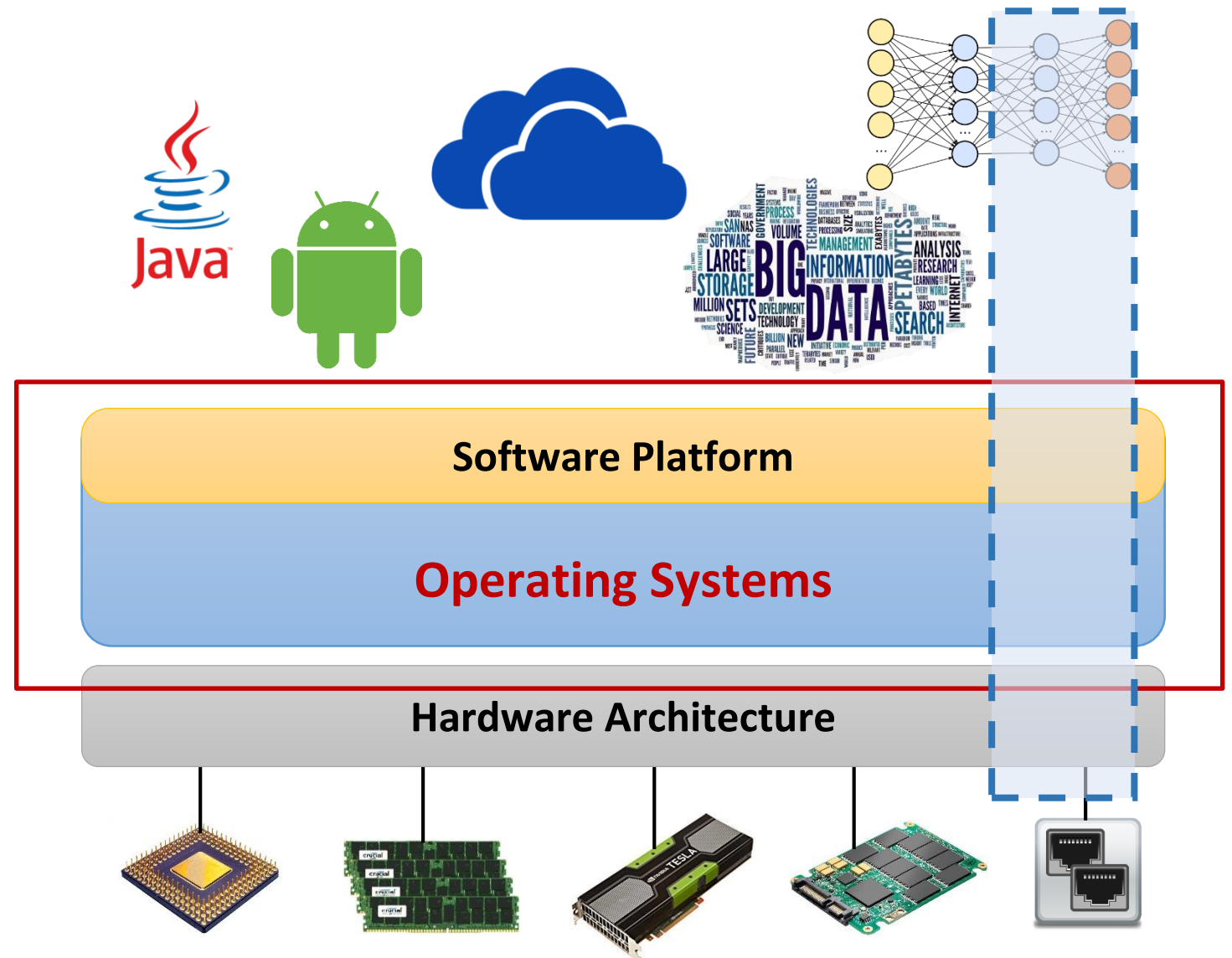
# Google TPU (Tensor Processing Unit)



# Computer Systems Research

- Architecture
- OS
- Compiler
- Network
- Database
- ...
- +
- Security

*“domain-specific computing”  
or “vertical optimization”*



Thank You!