CS 3410 Computer System Organization and Programming

Guest Lecture: I/O Devices

Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

Spring 2012

Agenda

I/O Device Examples, Organization, and Drivers

Programmed I/O vs. Memory-Mapped I/O

Polling-Based I/O vs. Interrupt-Based I/O

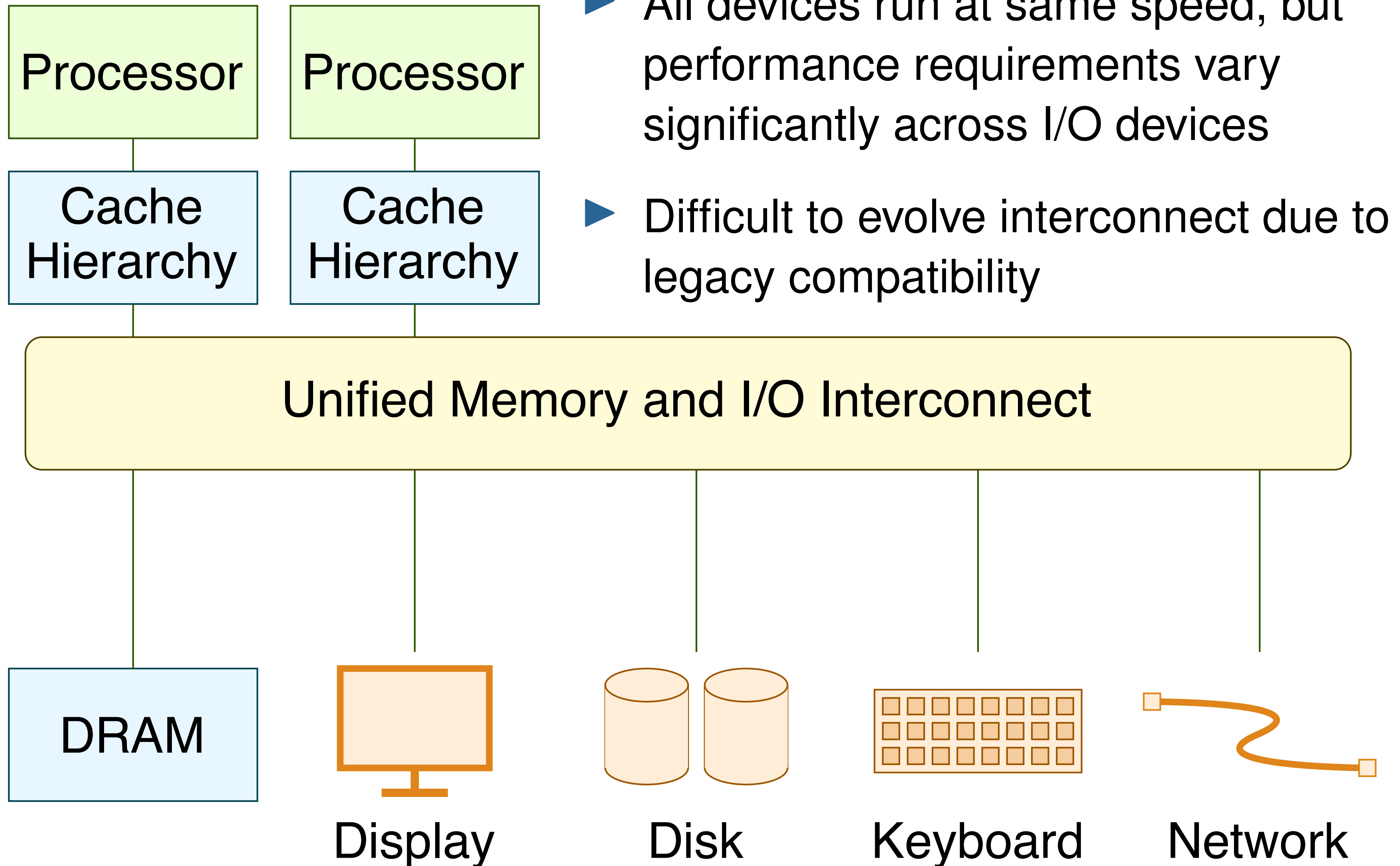
Direct-Memory Access

I/O Devices Enable Interacting with Environment

Device	Behavior	Partner	Data Rate (b/sec)
Keyboard	Input	Human	100
Mouse	Input	Human	3.8K
Sound Input	Input	Machine	3M
Voice Output	Output	Human	264K
Sound Output	Output	Human	8M
Laser Printer	Output	Human	3.2M
Graphics Display	Output	Human	800M–8G
Network/LAN	Input/Output	Machine	100M–10G
Network/Wireless LAN	Input/Output	Machine	11–54M
Optical Disk	Storage	Machine	5–120M
Flash Memory	Storage	Machine	32–200M
Magnetic Disk	Storage	Machine	800M–3G

Adapted from [Patterson'08]

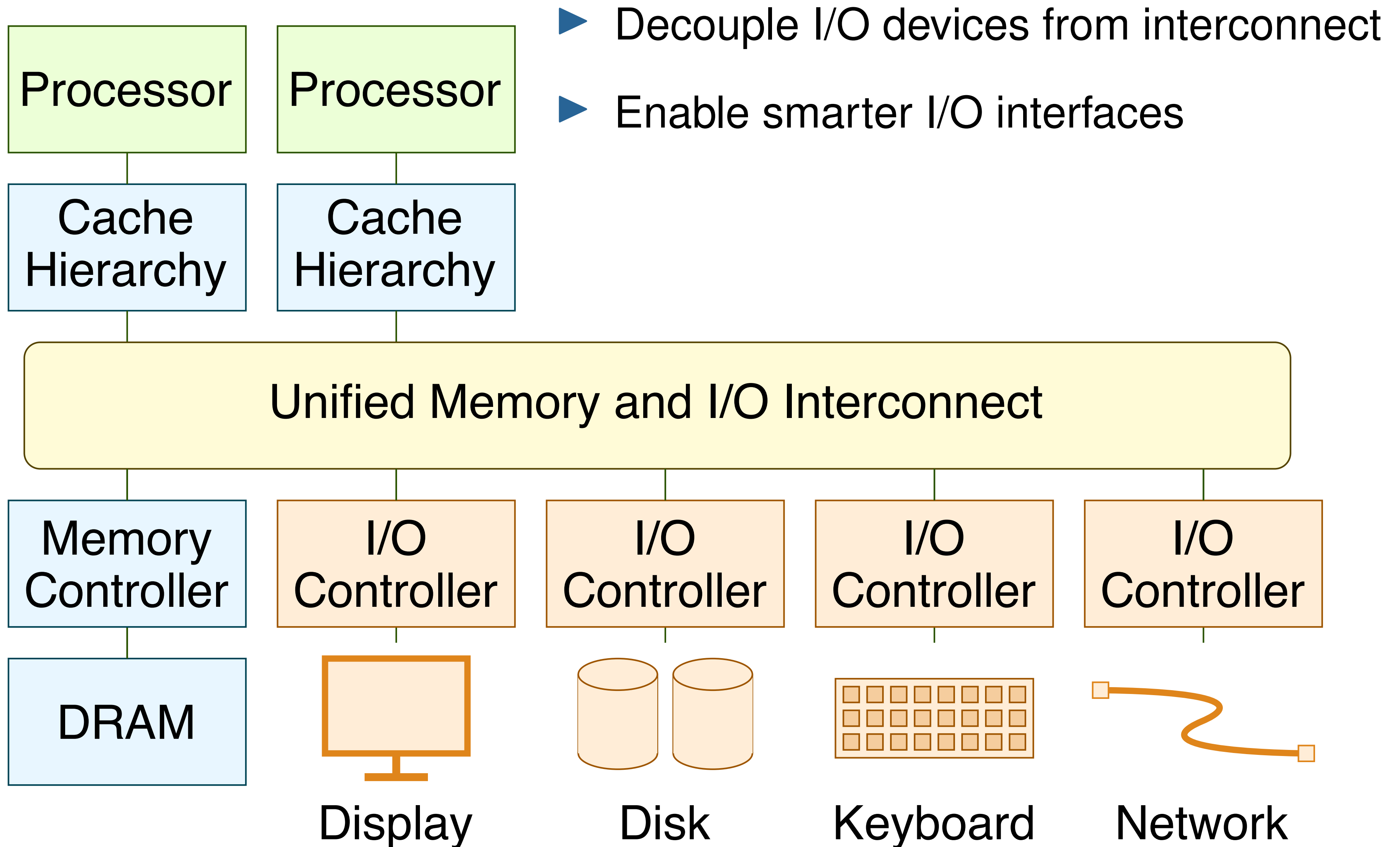
I/O Device Unified Interconnect



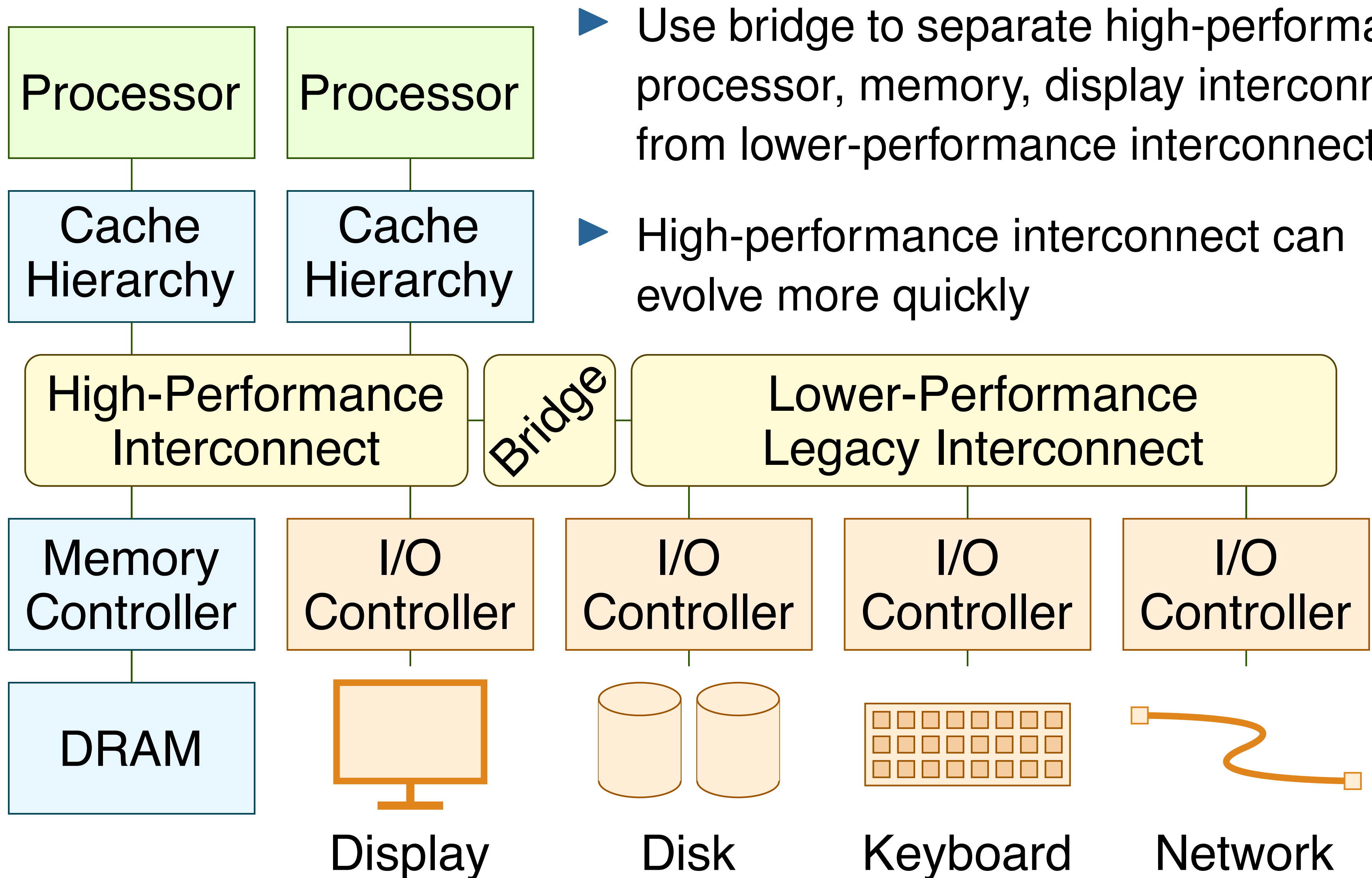
- All devices run at same speed, but performance requirements vary significantly across I/O devices
- Difficult to evolve interconnect due to legacy compatibility

I/O Device Controllers

- Decouple I/O devices from interconnect
- Enable smarter I/O interfaces

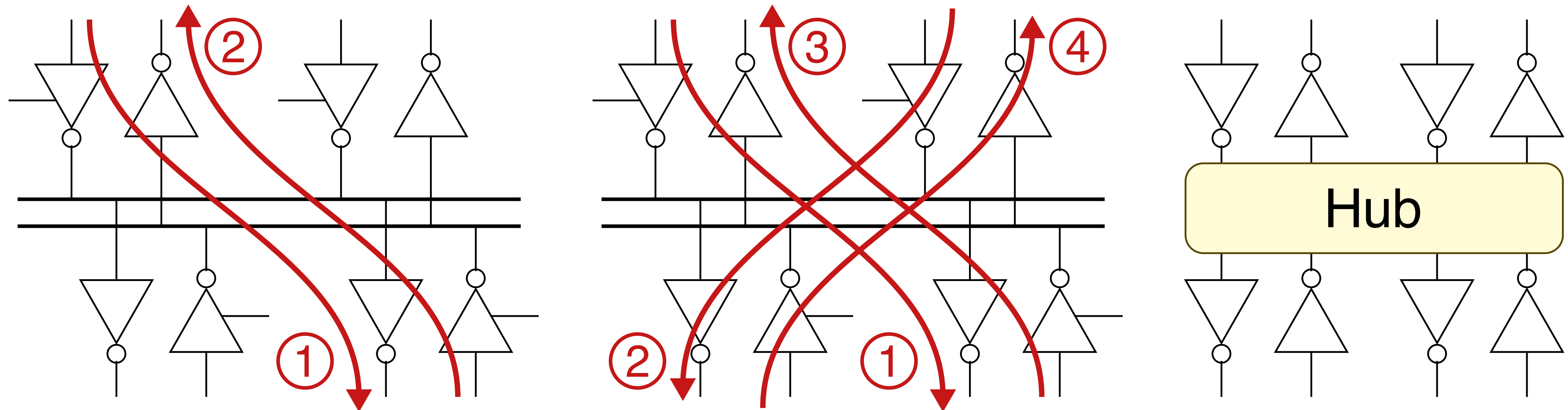


I/O Device Hierarchical Interconnect



- Use bridge to separate high-performance processor, memory, display interconnect from lower-performance interconnect
- High-performance interconnect can evolve more quickly

Bus vs. Point-to-Point Interconnect



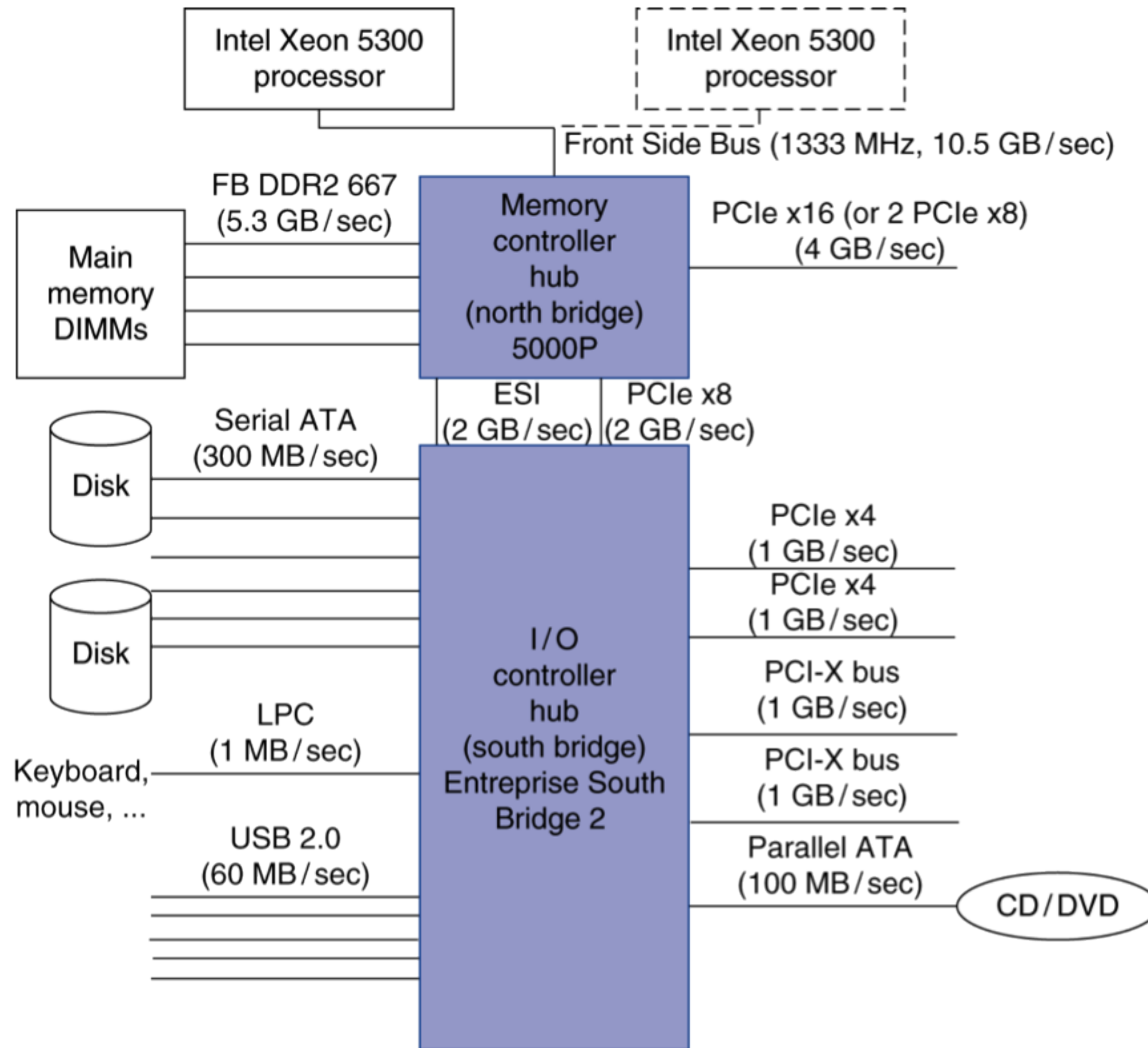
- Traditionally use slower wide parallel split-transaction busses
  - Simple to implement
  - More challenging to scale to long distances and high data rates
- More recently use high-speed narrow serial point-to-point channels
  - Much higher data-rates, distances, bandwidth density
  - More challenging to implement

Example Interconnects

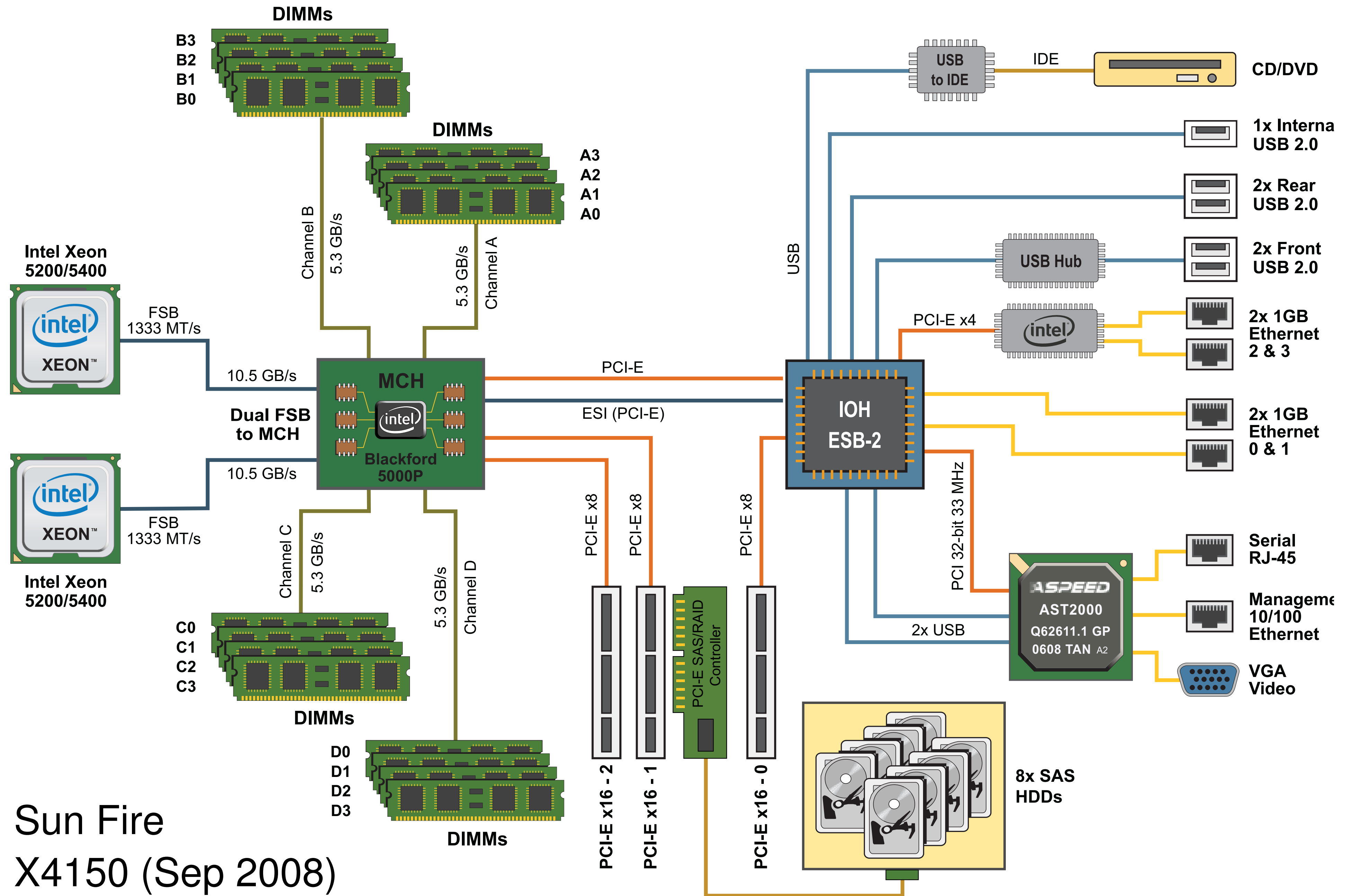
Name	Use	Dev Per Channel	Channel Width	Data Rate (B/sec)
Firewire 800	External	63	4	100M
USB 2.0	External	127	2	60M
Parallel ATA	Internal	1	16	133M
Serial ATA	Internal	1	4	300M
PCI 66MHz	Internal	1	32–64	533M
PCI Express v2.x	Internal	1	2–64	16G/dir
Hypertransport v3.1	Internal	1	2–64	25G/dir
QuickPath (QPI)	Internal	1	40	12G/dir

Adapted from [Patterson'08]

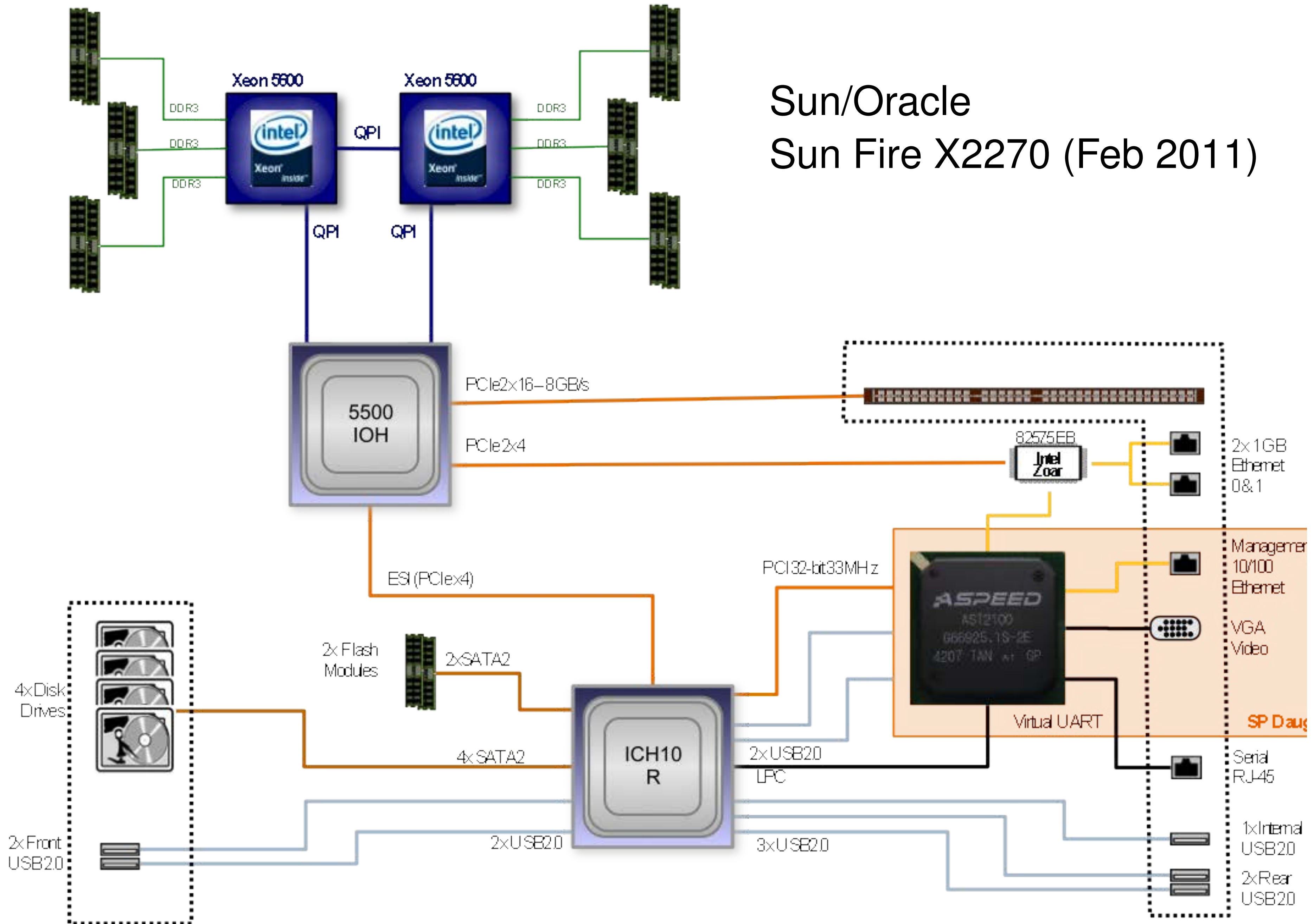
Traditional Hierarchical Bus-Based Interconnect



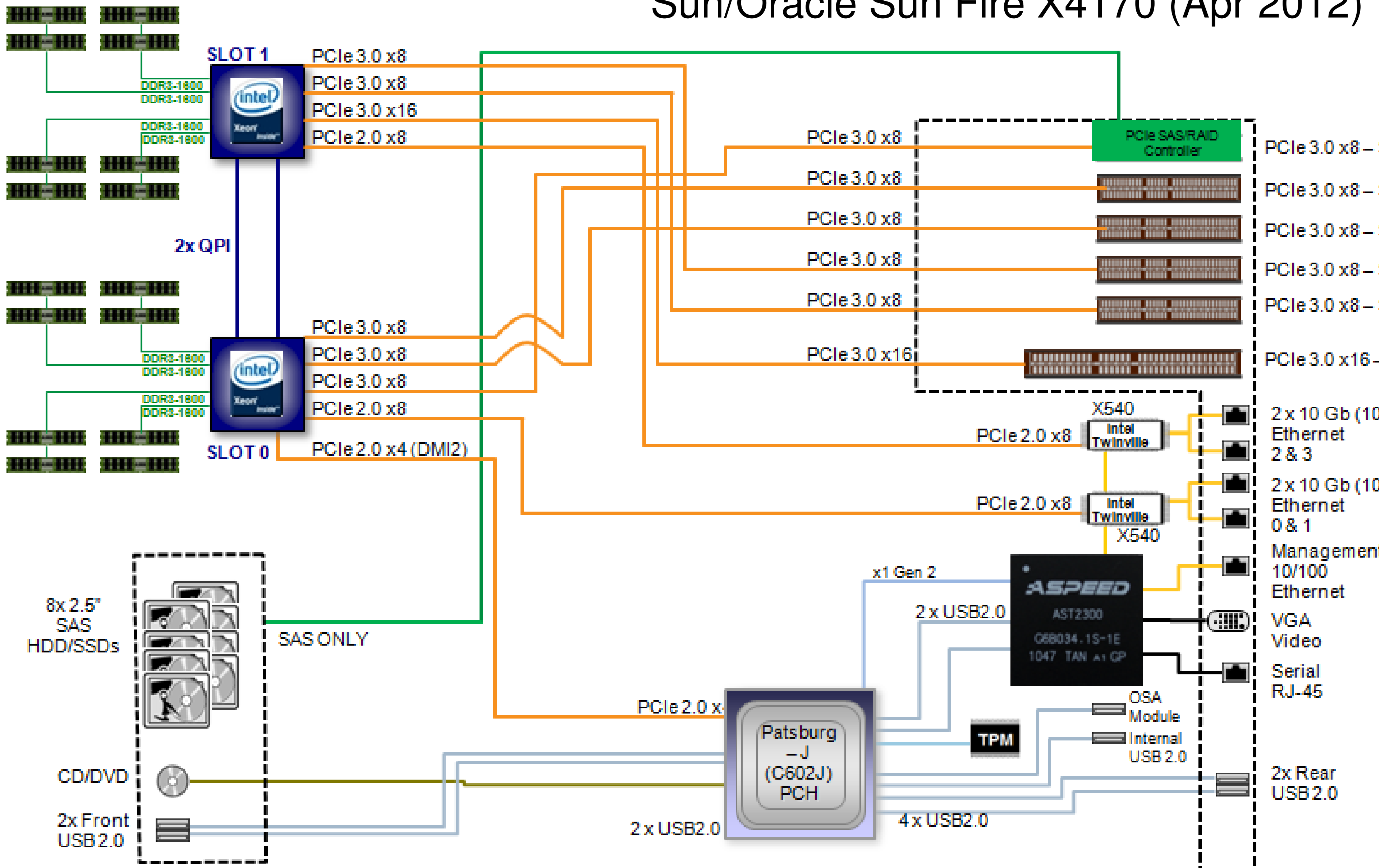
Adapted from [Patterson'08]

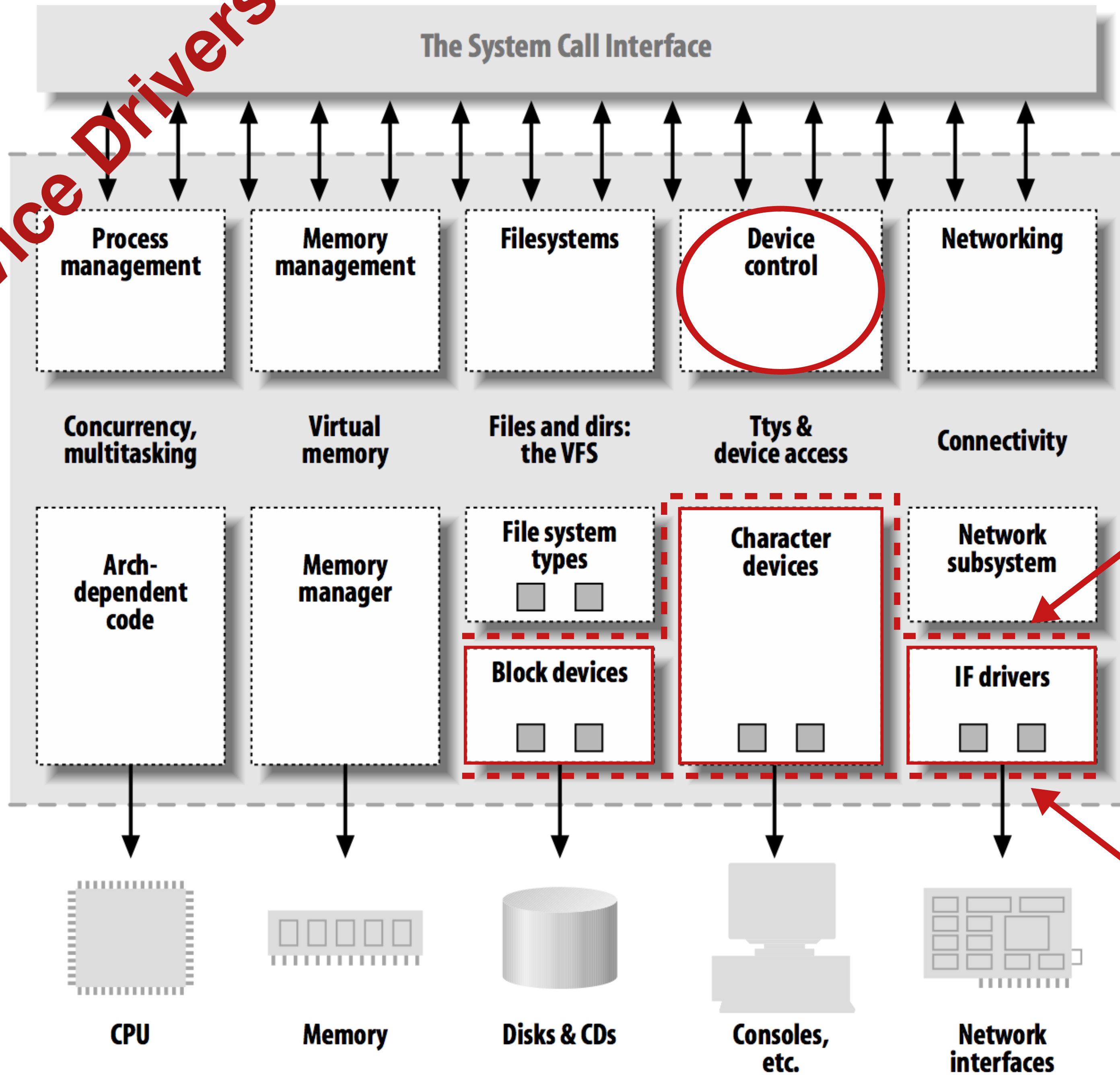


Sun Fire X4150 (Sep 2008)


Sun/Oracle
Sun Fire X2270 (Feb 2011)
Xeon 5500
Xeon 5500
QPI
QPI
QPI
DDR3
DDR3
DDR3
DDR3
DDR3
DDR3
5500
IOH
PCIe2x16 – 8GB/s
PCIe2x4
82575EB
Intel Zoar
2x 1GB Ethernet 0 & 1
ESI (PCIex4)
PCI 32-bit 33MHz
ASPEED
Management 10/100 Ethernet
VGA Video
Virtual UART
2x Flash Modules
2xSATA2
4x Disk Drives
4x SATA2
ICH10
R
2x USB2.0
LPC
Serial RJ-45
2x Front USB2.0
2x USB2.0
3x USB2.0
1x Internal USB2.0
2x Rear USB2.0

Sun/Oracle Sun Fire X4170 (Apr 2012)





I/O Device Driver Software Interface

I/O Device Driver Hardware Interface

Adapted from [Corbet'05]

I/O Device Driver Hardware Interface

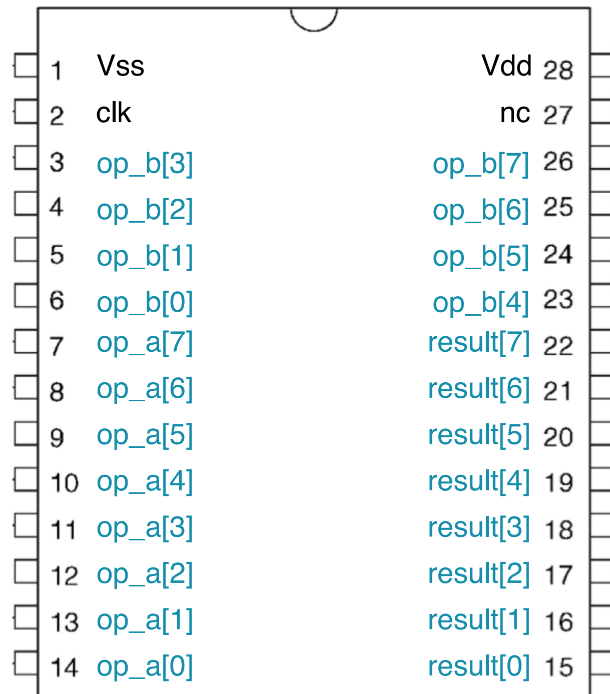
- Set of read-only or read/write device registers
- Status registers
  - Read to determine what device is doing, error codes, etc
- Data registers
  - Write: transfer data to a device
  - Read: read data from a device
- Command registers
  - Writing causes device to do something
- Example: AT Keyboard Device
  - 8-bit Status: parity error, input buf empty, output buf empty
  - 8-bit Command: 0xAA=”self test”, 0xAe=”enable kbd”, 0xED=”set LEDs”
  - 8-bit Data: scancode when reading, LED state when writing

I/O Device Driver Software Interface

- Set of methods to write/read data to/from device and control device
- Example: Linux Character Devices

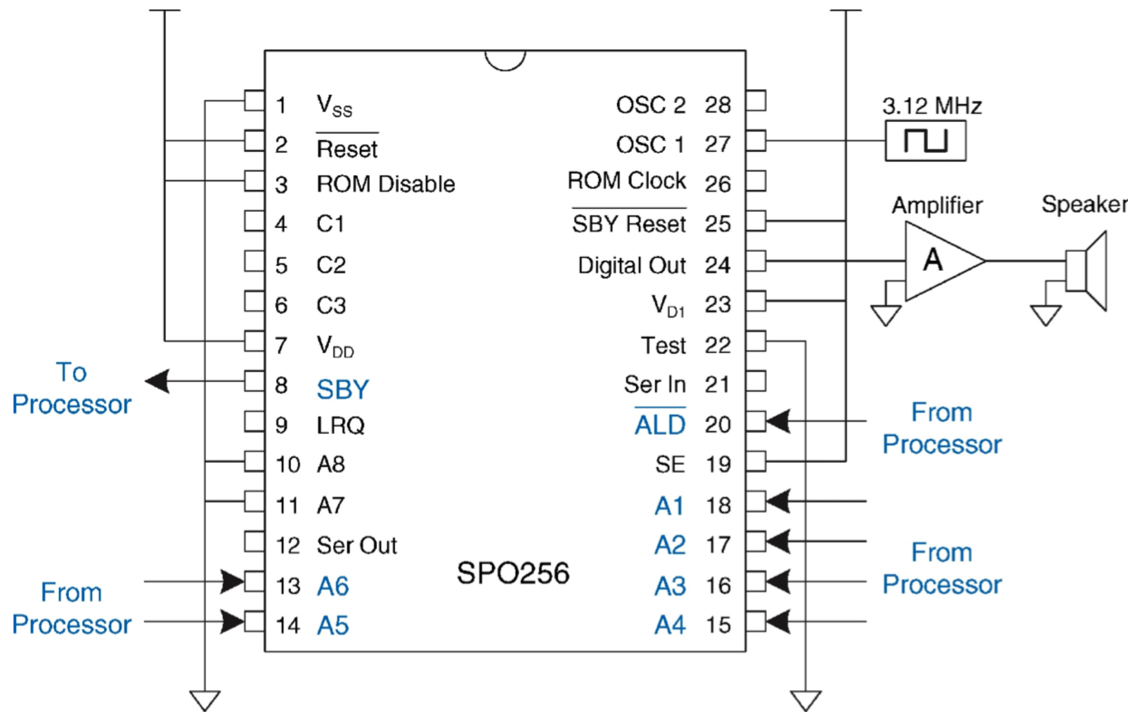
```
// Open a toy "echo" character device
int fd = open( "/dev/echo", O_RDWR );

// Write to the device
char write_buf[] = "Hello␣World!";
write( fd, write_buf, sizeof(write_buf) );

// Read from the device
char read_buf[32];
read( fd, read_buf, sizeof(read_buf) );

// Close the device
close( fd );

// Verify the result
assert( strcmp( write_buf, read_buf ) == 0 );
```

Example Device #1: Adder Chip



- Place and hold 8-bit data on op_a and op_b
- Read 8-bit result

Example Device #2: SPO256 Speech Synthesis Chip



- Wait until SBY high, means chip is ready to speak next allophone
- Write 6-bit allophone to A pins
- Toggle ALD signal low and then high
- Synthesis chip will then generate synthesized speech on digital output

Adapted from [Harris'07]

Example Device Organization

Adder Device Registers
- Data: op_a
- Data: op_b
- Data: result

Speech Device Registers
- Data: Allophone bits
- Command: ALD
- Status: SBY

Processor

Cache Hierarchy

Simple Single-Transaction Bus

Memory Controller

I/O Controller

I/O Controller

DRAM

Adder Device

Speech Device

Example Device Driver Interfaces

Adder Device

```
// Do calculation
uchar adder_calc( uchar op_a, uchar op_b );
```

SPO256 Speech Synthesis Device

```
// Synthesize speech for given allophones
void spo256_genspeech( uchar aph_buf[],
                       int aph_buf_size );
```

Agenda

I/O Device Examples,
Organization, and Drivers

Programmed I/O vs.
Memory-Mapped I/O

Polling-Based I/O vs.
Interrupt-Based I/O

Direct-Memory Access

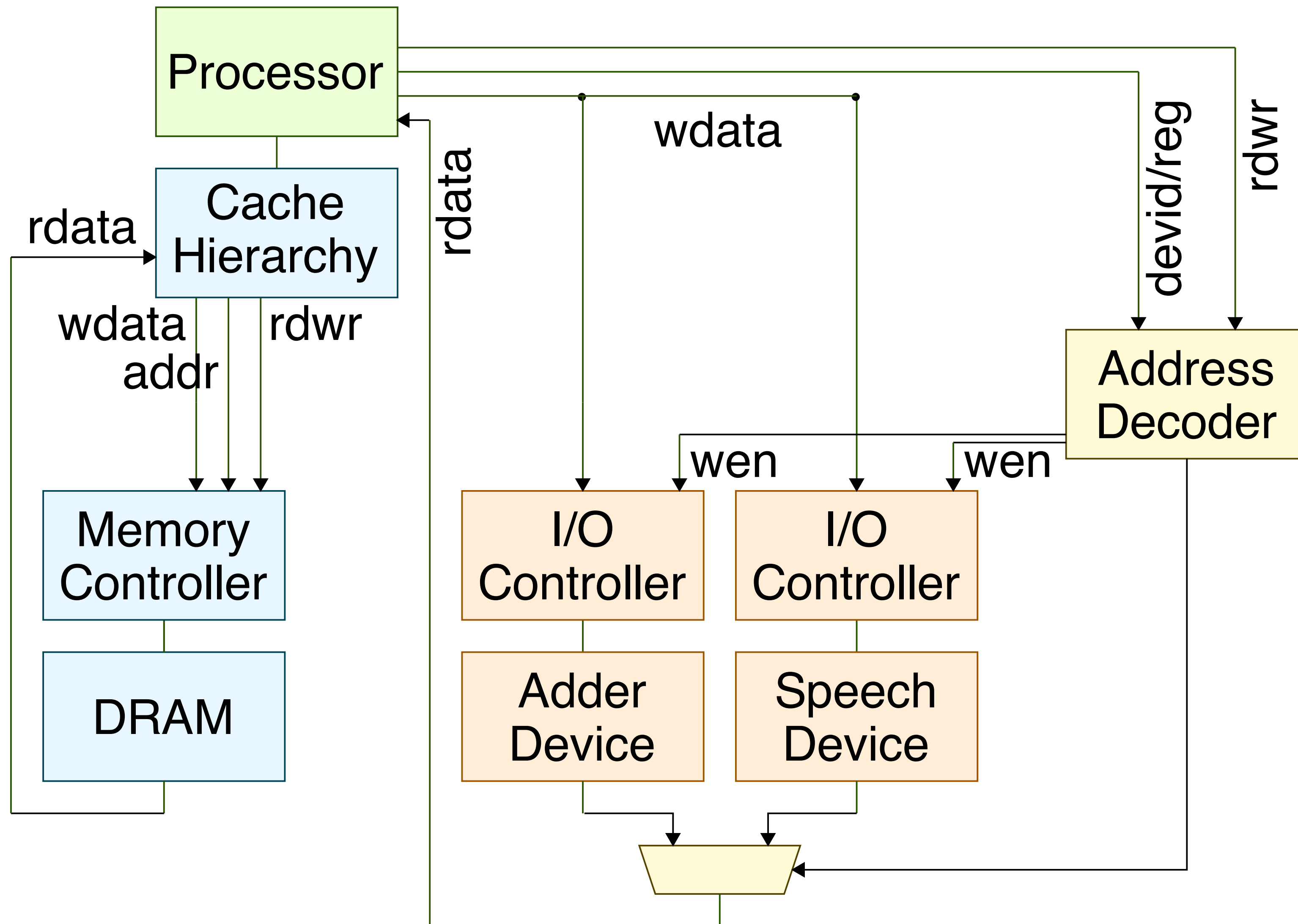
Programmed I/O

```
li   $r1, DEV_ADD_ID
li   $r2, DEV_ADD_REG_OPA
outb $r3, $r1, $r2
li   $r2, DEV_ADD_REG_OPB
outb $r4, $r1, $r2
li   $r2, DEV_ADD_REG_RES
inb  $r5, $r1, $r2
```

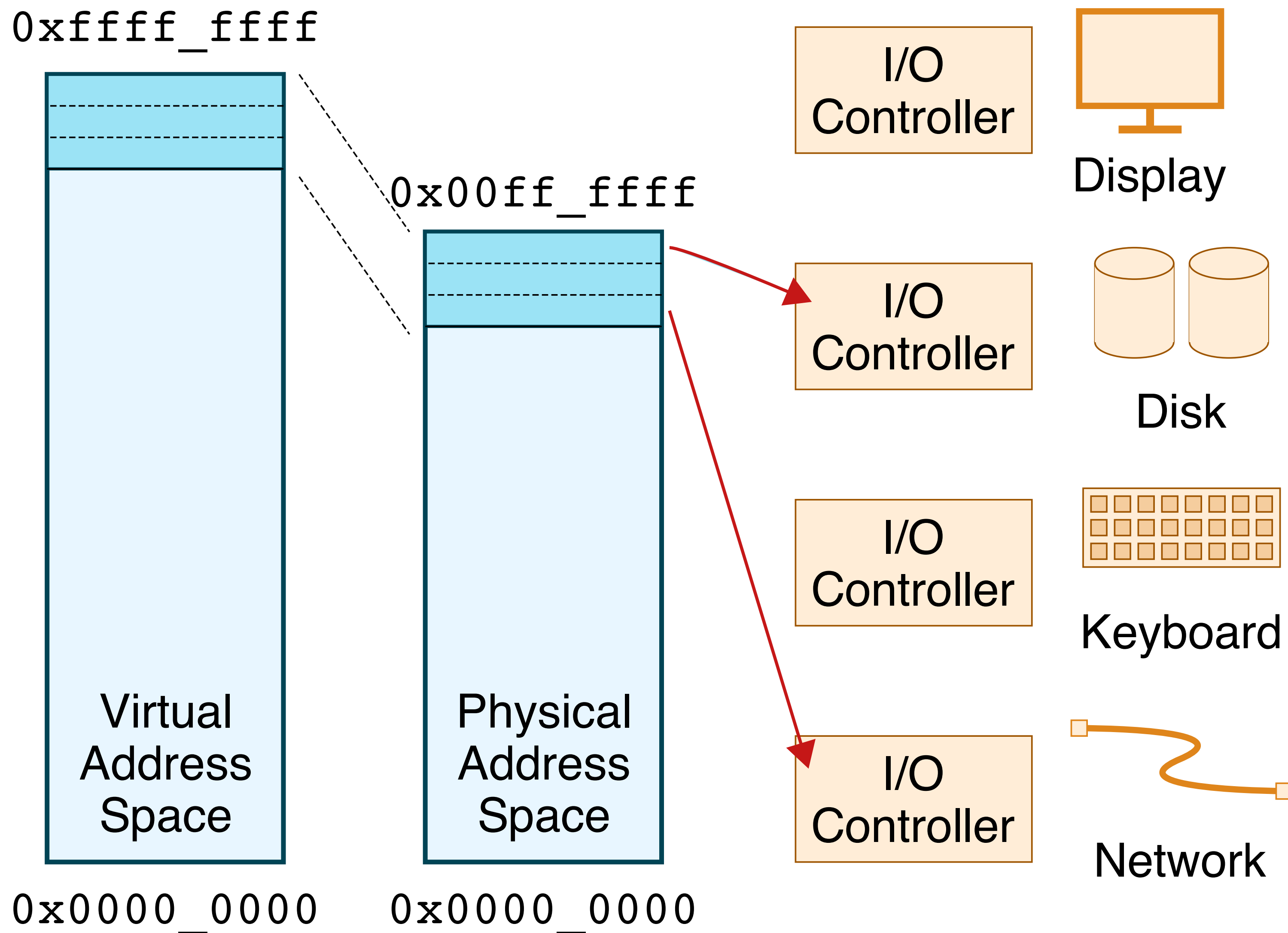
- Special instructions to read/write device registers
- Only allowed in kernel mode
- Used in IA32 instruction set

```
uchar adder_calc( uchar op_a, uchar op_b )
{
  outb( DEV_ADD_ID, DEV_ADD_REG_OPA, op_a );
  outb( DEV_ADD_ID, DEV_ADD_REG_OPB, op_b );
  return inb( DEV_ADD_ID, DEV_ADD_REG_RES );
}
```

Implementing Programmed I/O



Memory-Mapped I/O



Memory-Mapped I/O

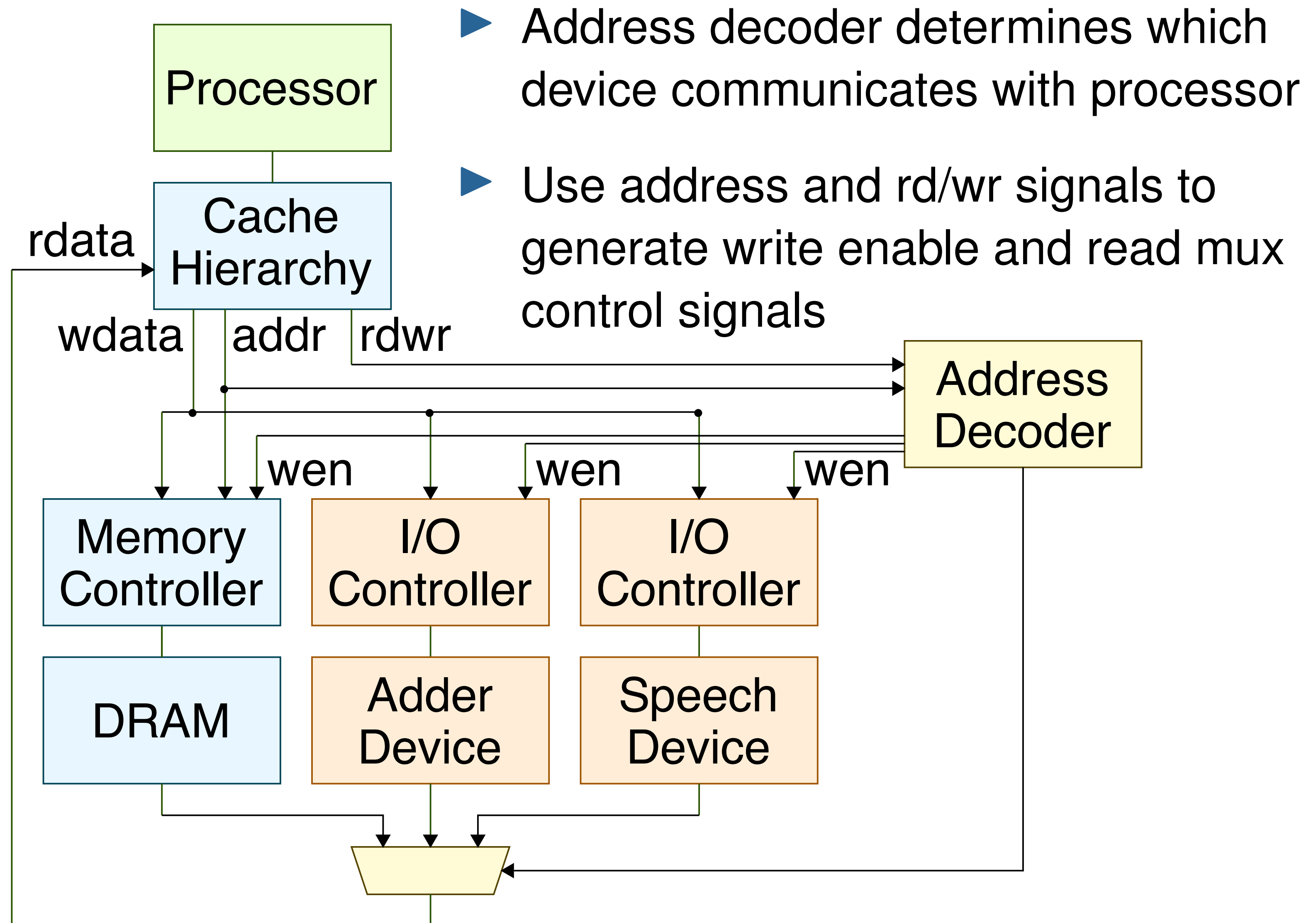
```
la   $r1, 0xffffe00
sw   $r3, 0($r1)
la   $r1, 0xffffe04
sw   $r4, 0($r1)
la   $r1, 0xffffe08
lw   $r5, 0($r1)
```

- Device registers mapped into address space
- Can use normal loads and stores
- Used in MIPS32 instruction set

```
volatile uint* adder_op_a   = (uint*) 0xffffe00;
volatile uint* adder_op_a   = (uint*) 0xffffe04;
volatile uint* adder_result = (uint*) 0xffffe08;

uchar adder_calc( uchar op_a, uchar op_b ) {
  *adder_op_a = op_a;
  *adder_op_a = op_a;
  return *adder_result;
}
```

Implementing Memory-Mapped I/O

- Address decoder determines which device communicates with processor
- Use address and rd/wr signals to generate write enable and read mux control signals

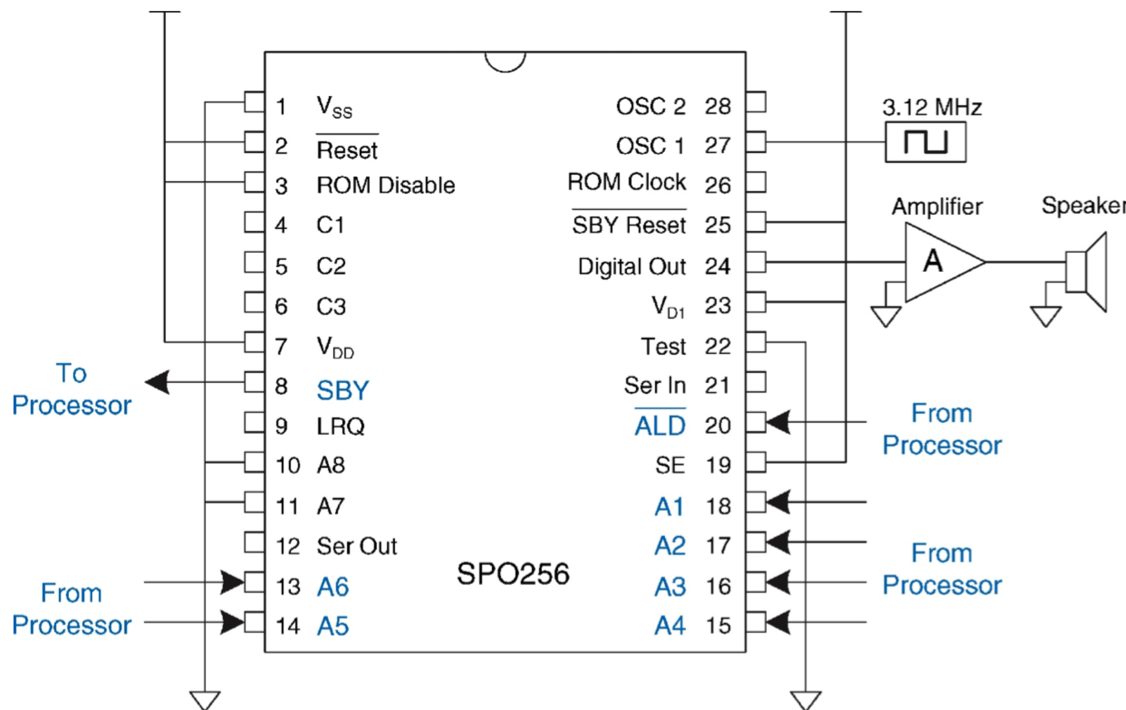


Comparing Programmed I/O vs. Memory-Mapped I/O

- Programmed I/O
  - Requires special instructions
  - Can require dedicated hardware interface to devices
  - Protection enforced via only allow kernel mode access to instructions
  - Virtualization can be difficult
- Memory-Mapped I/O
  - Re-uses standard load/store instructions
  - Re-uses standard memory hardware interface
  - Protection enforced with normal memory protection scheme
  - Virtualization enabled with normal memory virtualization scheme

Activity #1: Implement Polling Speech Device Driver

```
void spo256_genspeech( uchar aph_buf[],
                       int aph_buf_size );
```



- Wait until SBY high, means chip is ready to speak next allophone
- Write 6-bit allophone to A pins
- Toggle ALD signal low and then high
- Synthesis chip will then generate synthesized speech on digital output

Activity #1: Solution

```
volatile uint* spo256_abits = (uint*) 0xffffff00;
volatile uint* spo256_ald   = (uint*) 0xffffff04;
volatile uint* spo256_sby   = (uint*) 0xffffff08;

void spo256_genspeech( uchar aph_buf[], int aph_buf_size )
{
  // Ensure ALD is initially high
  *spo256_ald = 1;

  for ( int i = 0; i < aph_buf_size; i++ ) {

    // Wait for SBY to be high
    while ((*spo256_sby) == 0) {}

    // Write the allophone to the A pins
    *spo256_abits = aph_buf[i];

    // Initiate speech by toggling ALD
    *spo256_ald = 0;
    *spo256_ald = 1;
}
```

Activity #1: Solution (MIPS32 Assembly)

```
    li    $t0, 1              # constant one
    la    $t1, 0xffffff00     # pointer to device register A
    la    $t2, 0xffffff04     # pointer to device register ALD
    la    $t3, 0xffffff08     # pointer to device register SBY

    sw    $t0, 0($t2)         # *spo256_ald = 1

  loop:
    lw    $t4, 0($t3)         # $t3 = *spo256_sby
    beq   $t4, $0, loop       # loop until (*spo256_sby != 0)

    lw    $t4, 0($a0)         # $t5 = *aph_buf_ptr
    sw    $t4, 0($t1)         # *spo256_abits = *aph_buf_ptr
    sw    $0,  0($t2)         # *spo256_ald = 0
    sw    $t0, 0($t2)         # *spo256_ald = 1

    addiu $a0, $a0, 4         # aph_buf_ptr++
    addiu $a1, $a1, -1        # count--
    bgtz  $a1, loop           # loop until count == 0
```

Agenda

I/O Device Examples, Organization, and Drivers

Programmed I/O vs. Memory-Mapped I/O

Polling-Based I/O vs. Interrupt-Based I/O

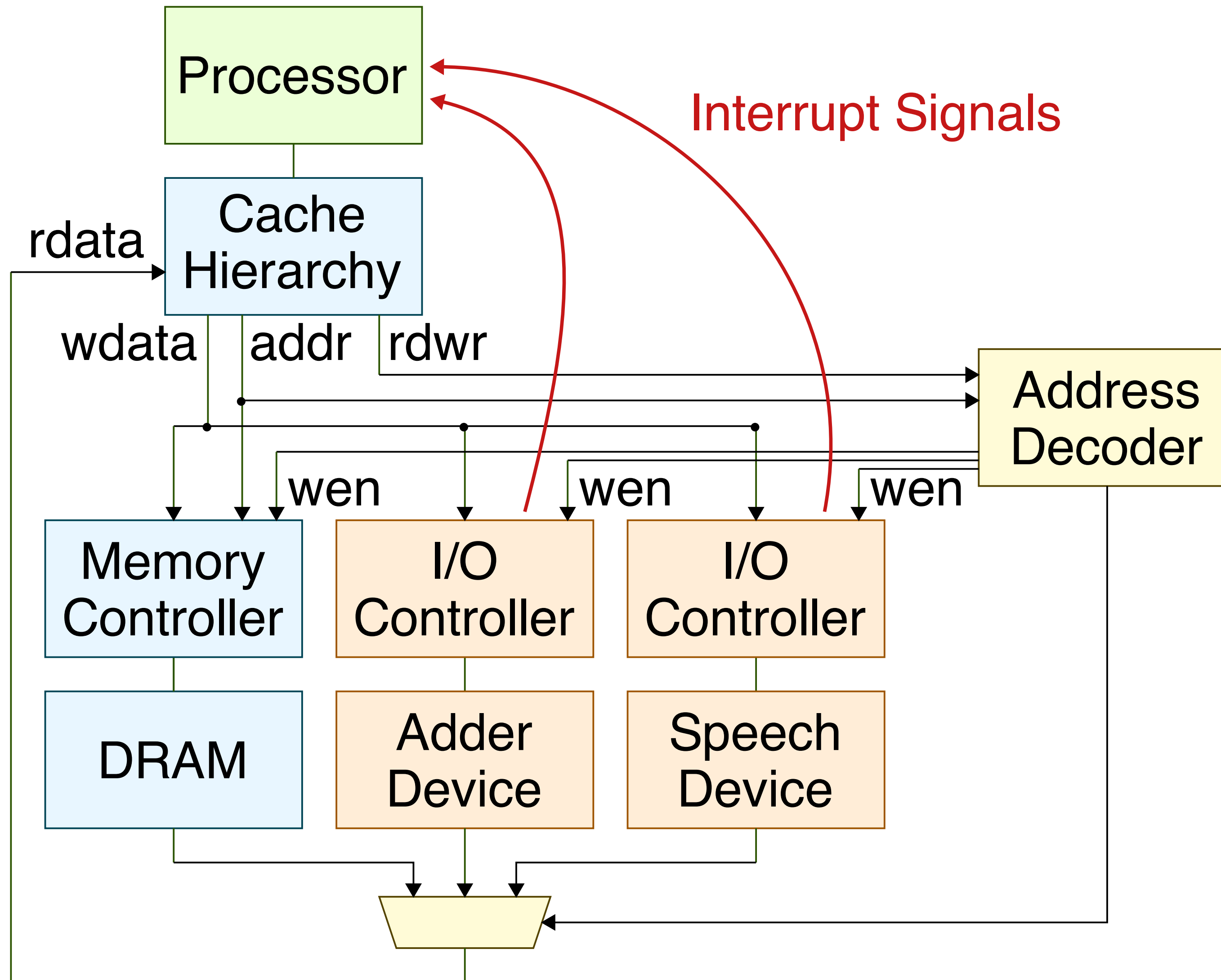
Direct-Memory Access

Polling-Based I/O

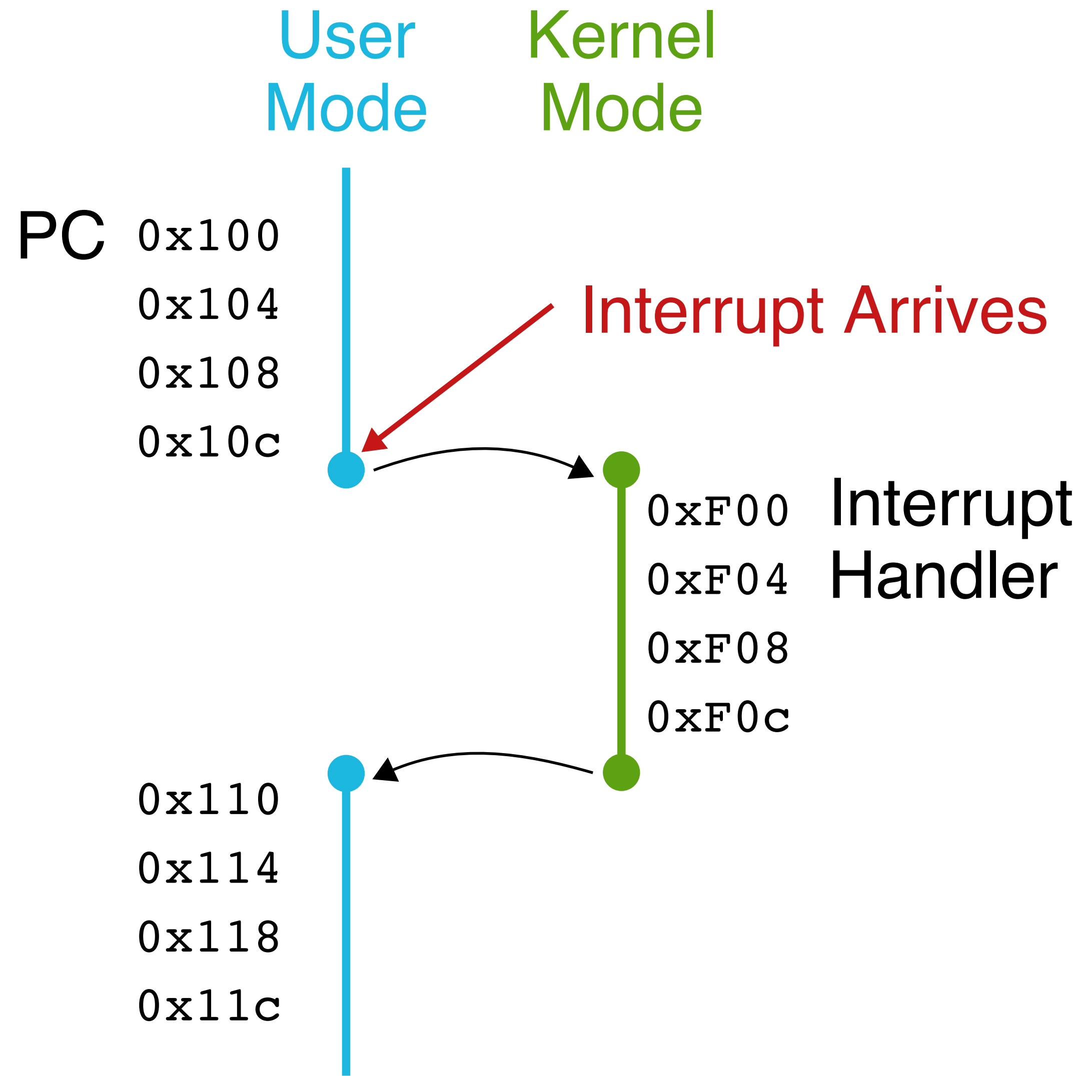
```
volatile uint* spo256_abits = (uint*) 0xffffff00;
volatile uint* spo256_ald   = (uint*) 0xffffff04;
volatile uint* spo256_sby   = (uint*) 0xffffff08;

void spo256_genspeech( uchar aph_buf[], int aph_buf_size )
{
  // Ensure ALD is initially high
  *spo256_ald = 1;

  for ( int i = 0; i < aph_buf_size; i++ ) {

    // Wait for SBY to be high
    while ((*spo256_sby) == 0) {}

    // Write the allophone ...
    *spo256_abits = aph_buf[i];

    // Initiate speech by ...
    *spo256_ald = 0;
    *spo256_ald = 1;
}
```

While loop is example of polling, wastes processor resources (performance, energy)

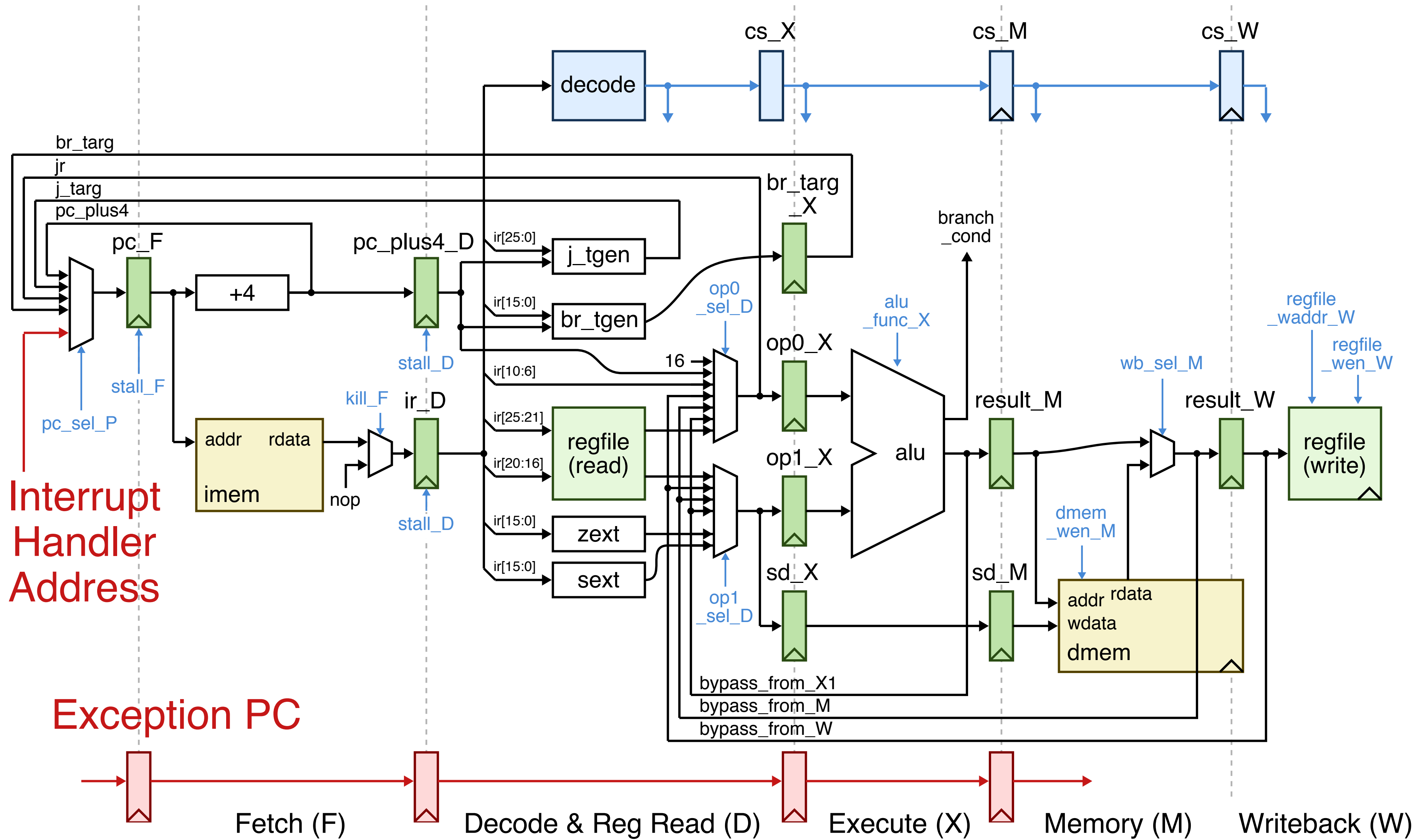
Implementing Interrupt-Based I/O



Implementing Interrupt-Based I/O



Implementing Interrupt-Based I/O

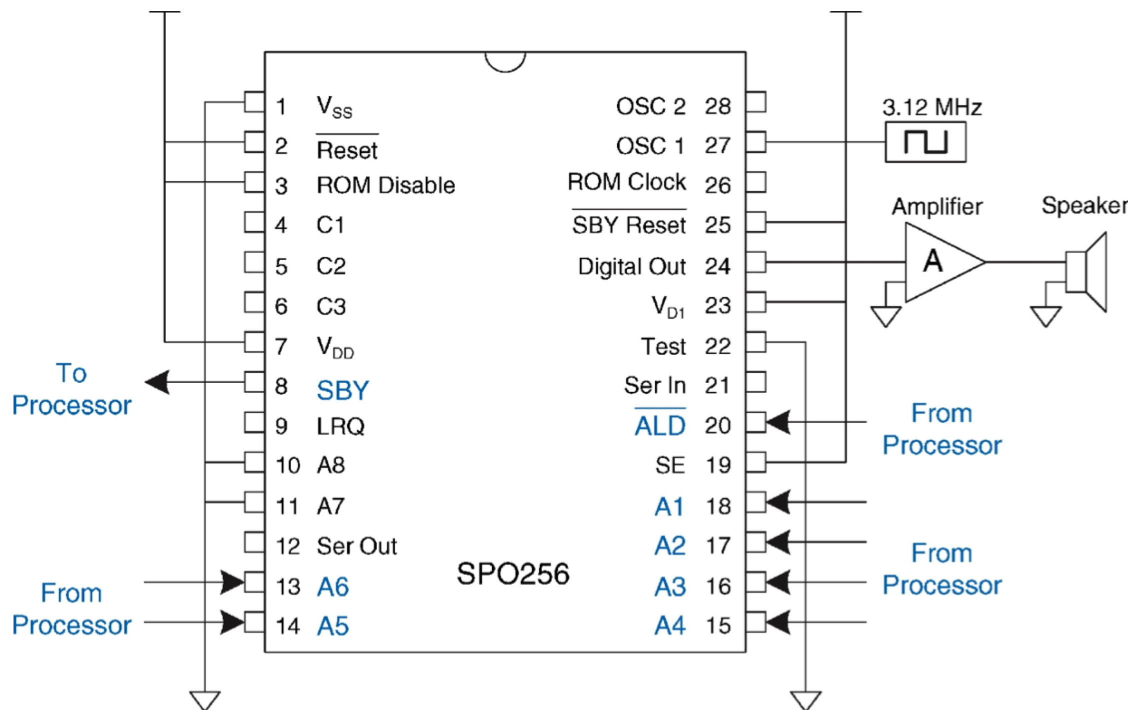


Comparing Polling-Based I/O vs. Interrupt-Based I/O

- Polling-Based I/O
  - Simple to implement
  - Predictable performance helpful for real-time applications
  - Can consume significant performance and energy resources while waiting
- Interrupt-Based I/O
  - More complicated to implement
  - More efficient in most cases

Activity #2: Implement Interrupt Speech Device Driver

```
void spo256_genspeech( uchar aph_buf[],
                       int aph_buf_size );
```



- Wait until SBY high, means chip is ready to speak next allophone
- Write 6-bit allophone to A pins
- Toggle ALD signal low and then high
- Synthesis chip will then generate synthesized speech on digital output

Activity #2: Solution (genspeech function)

```
// Assume device registers memory mapped as before
uchar* spo256_aph_buf[SPO256_APH_BUF_MAX_SIZE];
int    spo256_aph_buf_idx  = 1;
int    spo256_aph_buf_size = 0;

void spo256_genspeech( uchar aph_buf[], int aph_buf_size )
{
  // If currently handling a different request wait;
  // could enqueue req to enable multiple reqs in flight
  while ( spo256_aph_buf_size > 0 ) {}

  // Ensure ALD is initially low
  *spo256_ald = 1;

  // continued on next slide ...
```

Activity #2: Solution (genspeech function)

```
  // Handle first allophone
  if ( aph_buf_size > 0 ) {

    // Write the first allophone to the A pins
    *spo256_abits = aph_buf[0];

    // Initiate speech by toggling ALD
    *spo256_ald = 0;
    *spo256_ald = 1;
  }

  // Copy allophone buf to global buf for int handler
  if ( aph_buf_size > 1 ) {

    // Make sure space in global allophone buffer
    assert( aph_buf_size < SPO256_APH_BUF_MAX_SIZE );

    // Do copy and update global size
    memcopy( spo256_aph_buf, aph_buf, aph_buf_size );
    spo256_aph_buf_size = aph_buf_size;
  }
}
```

Activity #2: Solution (interrupt handler)

```
// Assume device registers memory mapped as before
uchar* spo256_aph_buf[SPO256_APH_BUF_MAX_SIZE];
int    spo256_aph_buf_idx  = 1;
int    spo256_aph_buf_size = 0;

void spo256_handle_interrupt()
{
  // Write the allophone to the A pins
  *spo256_abits = spo256_aph_buf[spo256_aph_buf_idx];

  // Initiate speech by toggling ALD
  *spo256_ald = 0;
  *spo256_ald = 1;

  // Increment index
  spo256_aph_buf_idx++;

  // Check if we are finished
  if ( spo256_aph_buf_idx == spo256_aph_buf_size )
    spo256_aph_buf_size = 0;
}
```

Agenda

I/O Device Examples, Organization, and Drivers

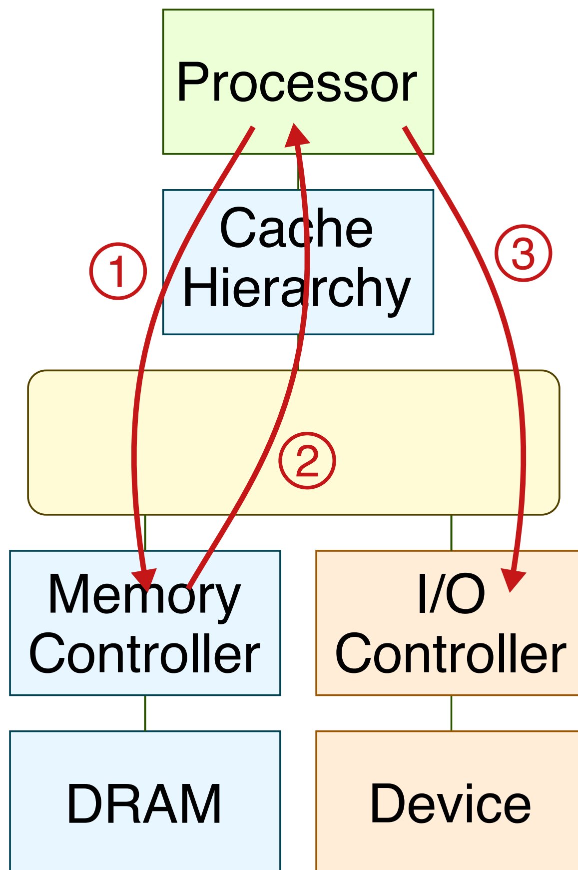
Programmed I/O vs. Memory-Mapped I/O

Polling-Based I/O vs. Interrupt-Based I/O

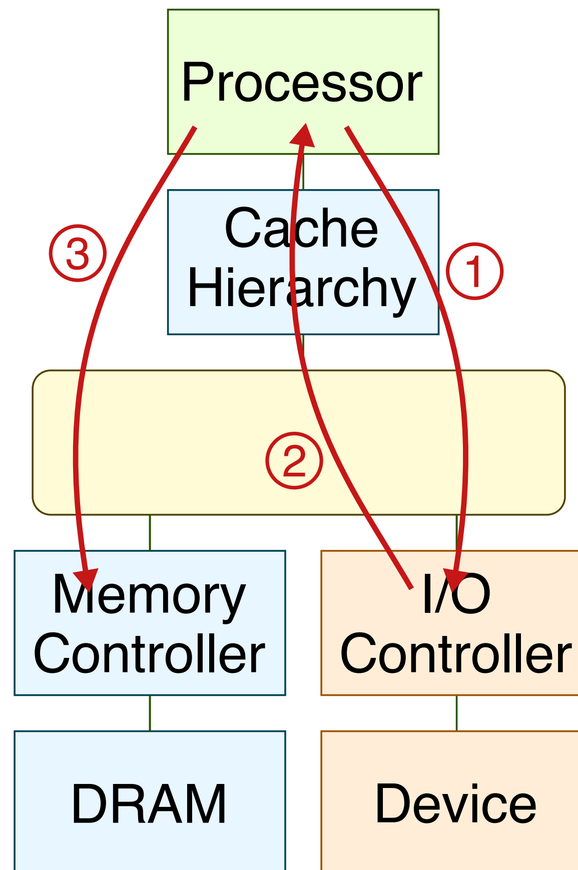
Direct-Memory Access

Direct-Memory Access

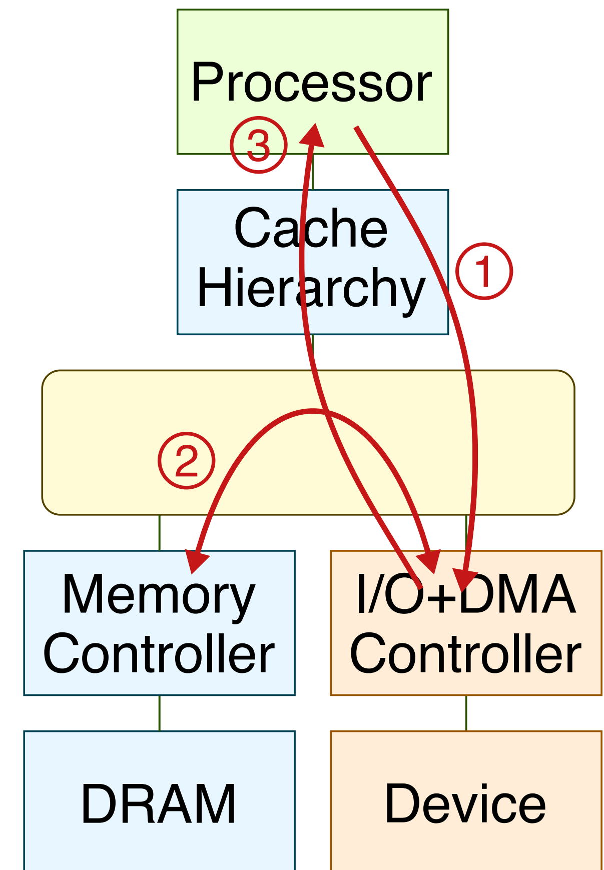
Indirect Memory to Device



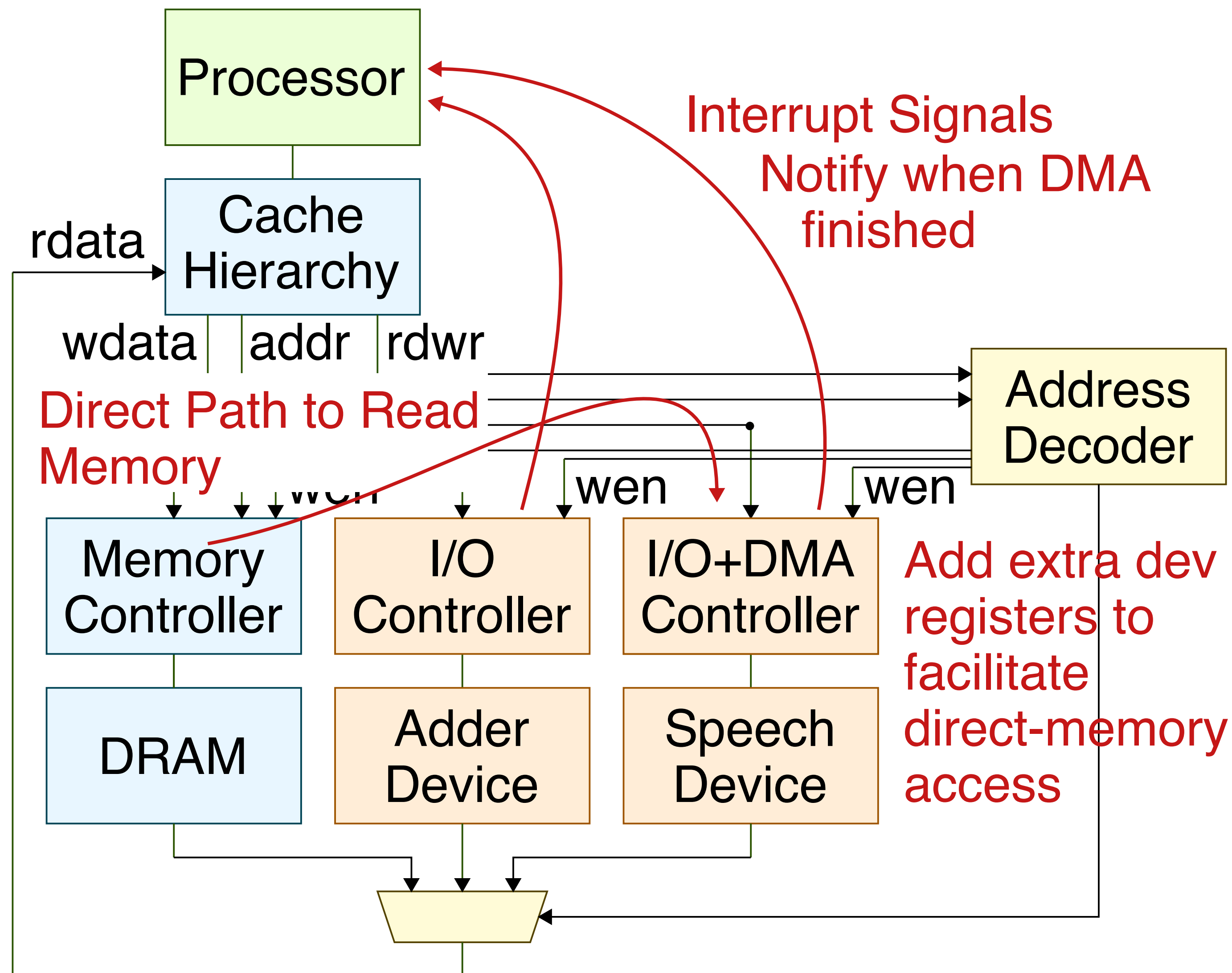
Indirect Device to Memory



Direct Memory Access



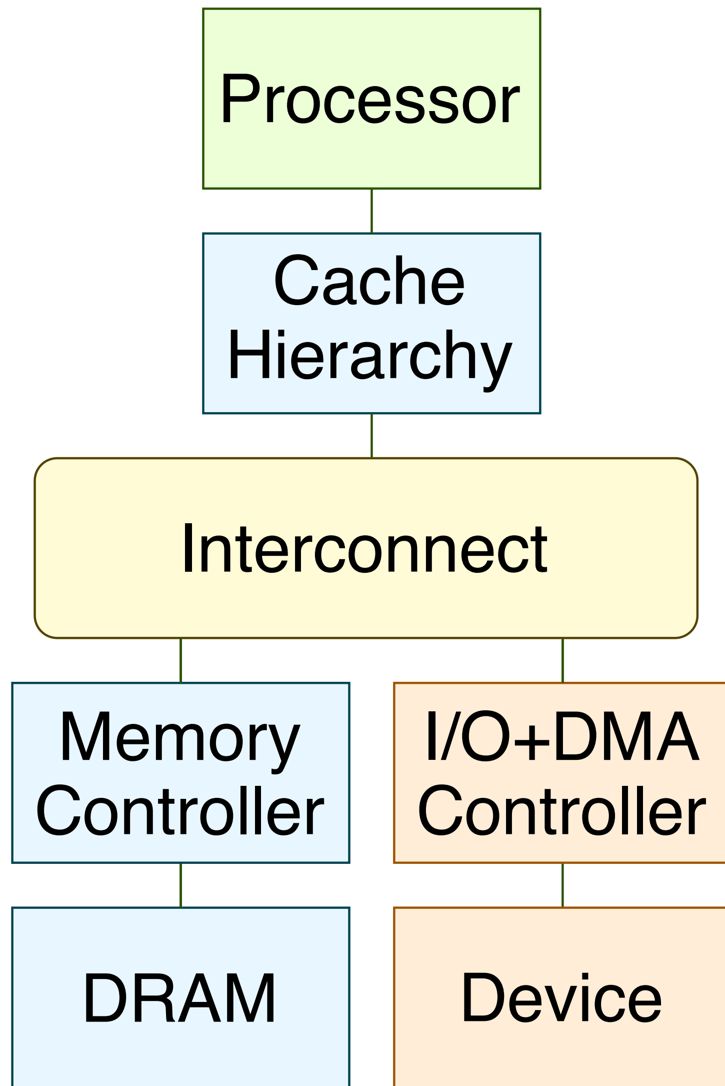
Implementing Direct-Memory Access



Speech Device with DMA Controller

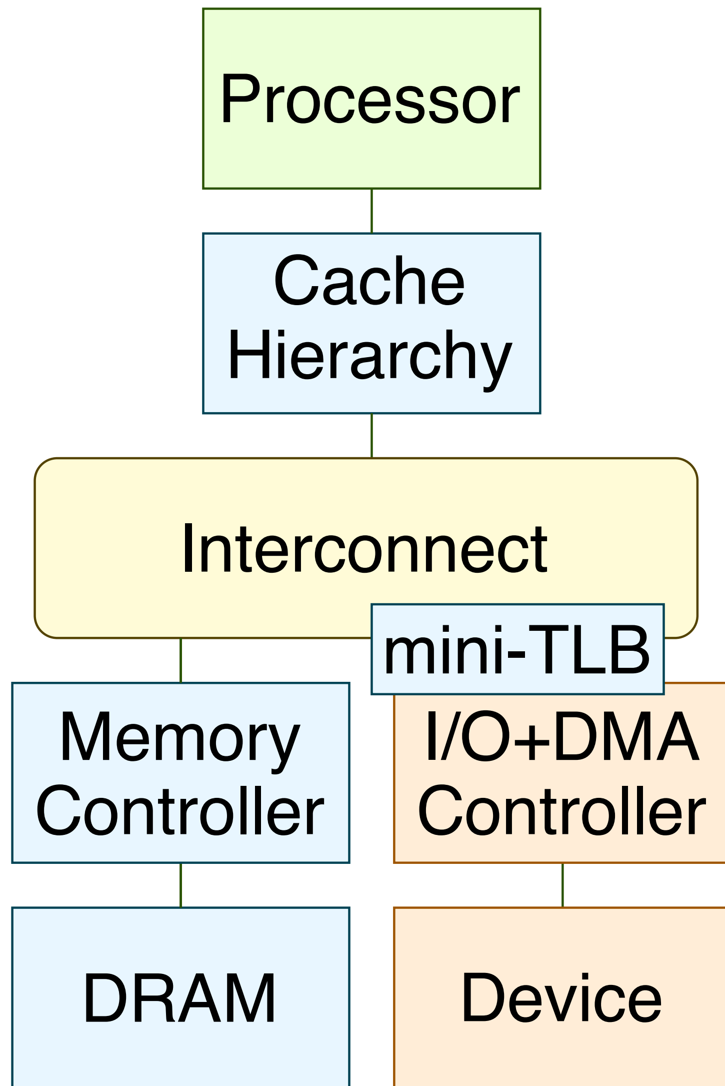
```
volatile uint* spo256_abits     = (uint*) 0xffffff00;
volatile uint* spo256_ald       = (uint*) 0xffffff04;
volatile uint* spo256_sby       = (uint*) 0xffffff08;

volatile uint* spo256_dma_addr  = (uint*) 0xffffff0c;
volatile uint* spo256_dma_len   = (uint*) 0xffffff10;
volatile uint* spo256_dma_start = (uint*) 0xffffff14;

void spo256_genspeech( uchar aph_buf[], int aph_buf_size )
{
  // Allocate DMA buffer
  uchar* dma_buf = dma_alloc( aph_buf_size );

  // Copy allophone buffer into DMA buffer
  memcopy( dma_buf, aph_buf, aph_buf_size );

  // Initialize DMA transfer
  *spo256_dma_addr  = dma_buf;
  *spo256_dma_len   = aph_buf_size;
  *spo256_dma_start = 1;
}
```

Direct-Memory Access Issue #1: Addressing



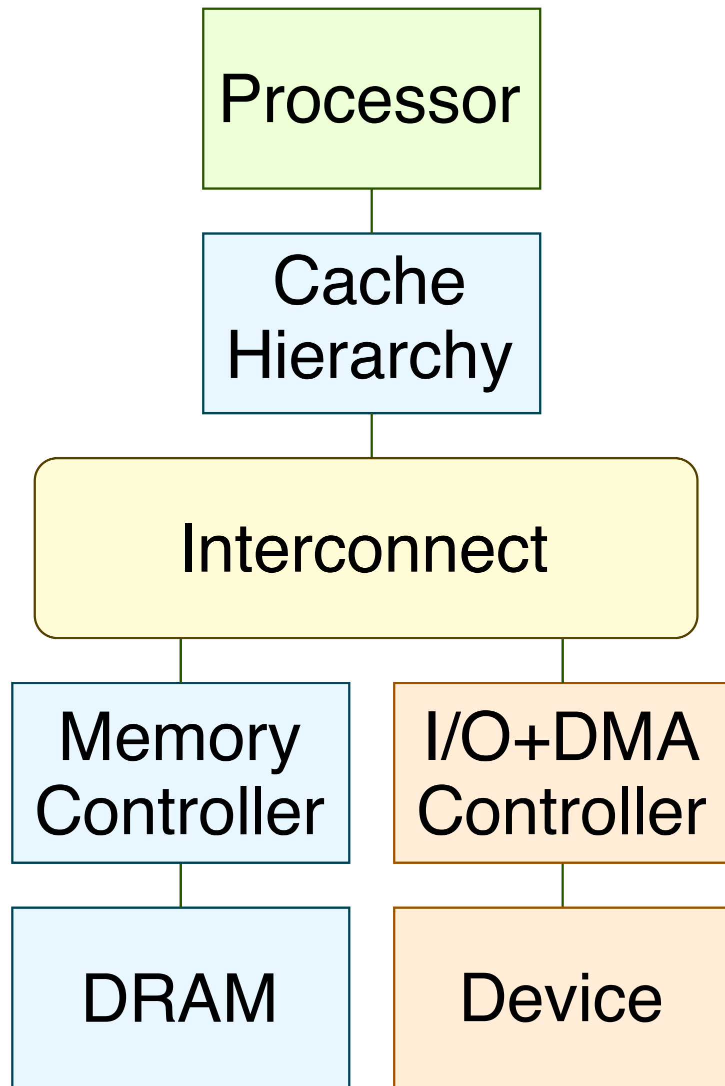
- Problem: Programs use virtual addresses, while memory controller uses physical addresses
- Solution: DMA uses physical addresses
  - OS uses physical addresses when setting up direct-memory transfer
  - OS allocates continuous physical pages for direct-memory transfer
  - OR: OS splits transfer into page-sized chunks (usually need ability to queue up multiple transfers)

Direct-Memory Access Issue #1: Addressing



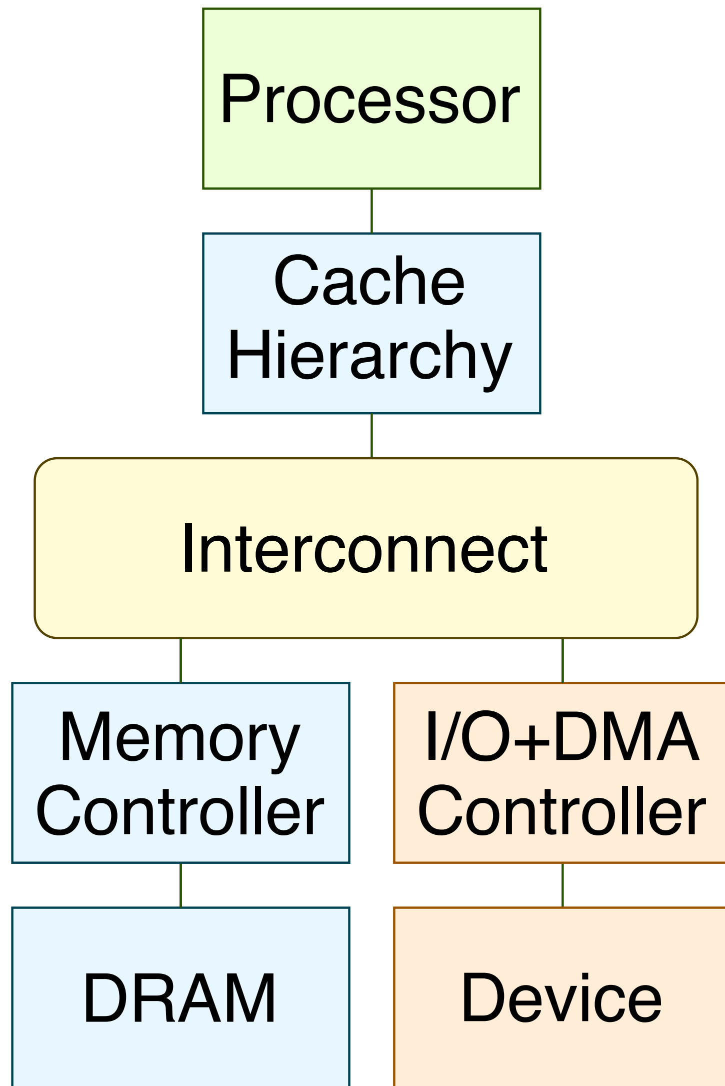
- Problem: Programs use virtual addresses, while memory controller uses physical addresses
- Solution: DMA uses virtual addresses
  - DMA controller needs its own “mini-TLB”
  - OS sets up “mini-TLB” before starting direct-memory transfer

Direct-Memory Access Issue #2: Virtual Memory



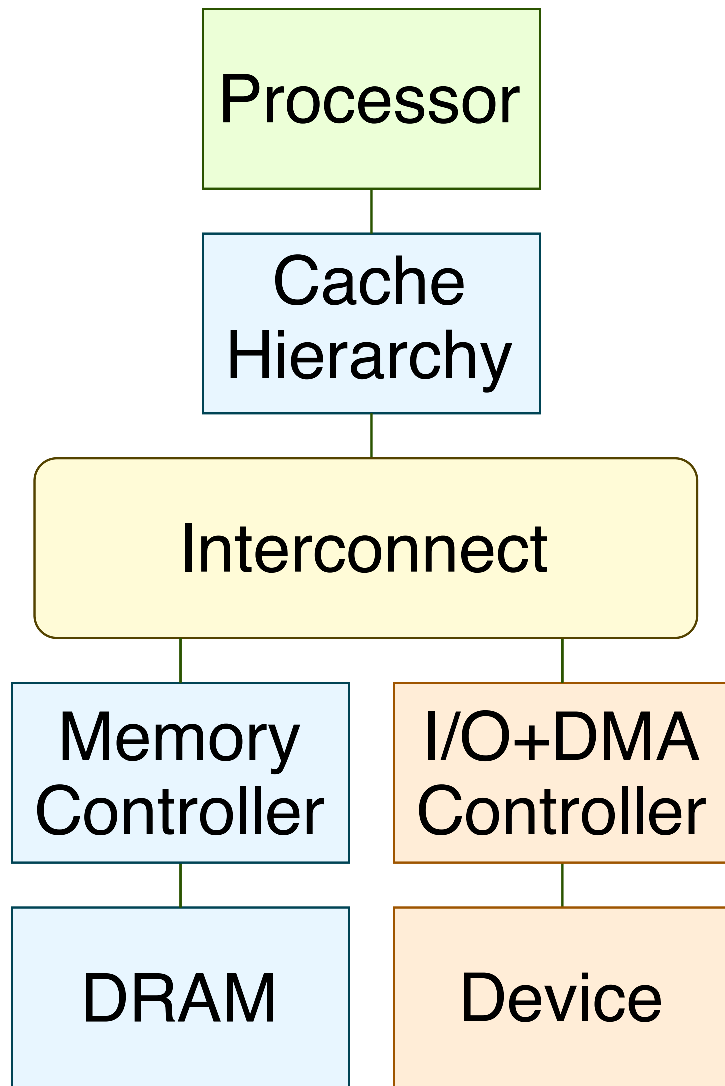
- Problem: DMA source/destination page may get swapped out before direct-memory transfer is complete
- Solution: OS pins the page before initiating direct-memory transfer

Direct-Memory Access Issue #3: Cache Coherence



- Problem: DMA-related data could be cached in L1/L2
  - DMA to memory: cache is now stale
  - Memory to DMA: device gets stale copy
- Solution: Software enforced coherence
  - OS flushes some/all cache before DMA begins
  - OR: don't touch pages during DMA
  - OR: mark pages as uncacheable in page table entries

Direct-Memory Access Issue #3: Cache Coherence



- Problem: DMA-related data could be cached in L1/L2
  - DMA to memory: cache is now stale
  - Memory to DMA: device gets stale copy
- Solution: Hardware enforced coherence
  - Cache listens on bus and determines when to invalidate or writeback copies
  - Can piggyback on coherence needed for multi-processors systems

Take-Away Points

- Diverse I/O devices require hierarchical interconnect which is more recently transitioning to point-to-point topologies
- Memory-mapped I/O is an elegant technique to read/write device registers with standard load/stores
- Interrupt-based I/O avoids the wasted work in polling-based I/O and is usually more efficient
- Modern systems combine memory-mapped I/O, interrupt-based I/O, and direct-memory access to create sophisticated I/O device subsystems

Acknowledgments

- [Corbet'05] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux Device Drivers," 3rd ed, O'Reilly Media, 2005.
- [Harris'07] D. Harris and S. Harris, "Digital Design and Computer Architecture," Morgan Kaufmann, 2007.
- [Patterson'08] D.A. Patterson and J.L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface," 4th ed, Morgan Kaufmann, 2008.