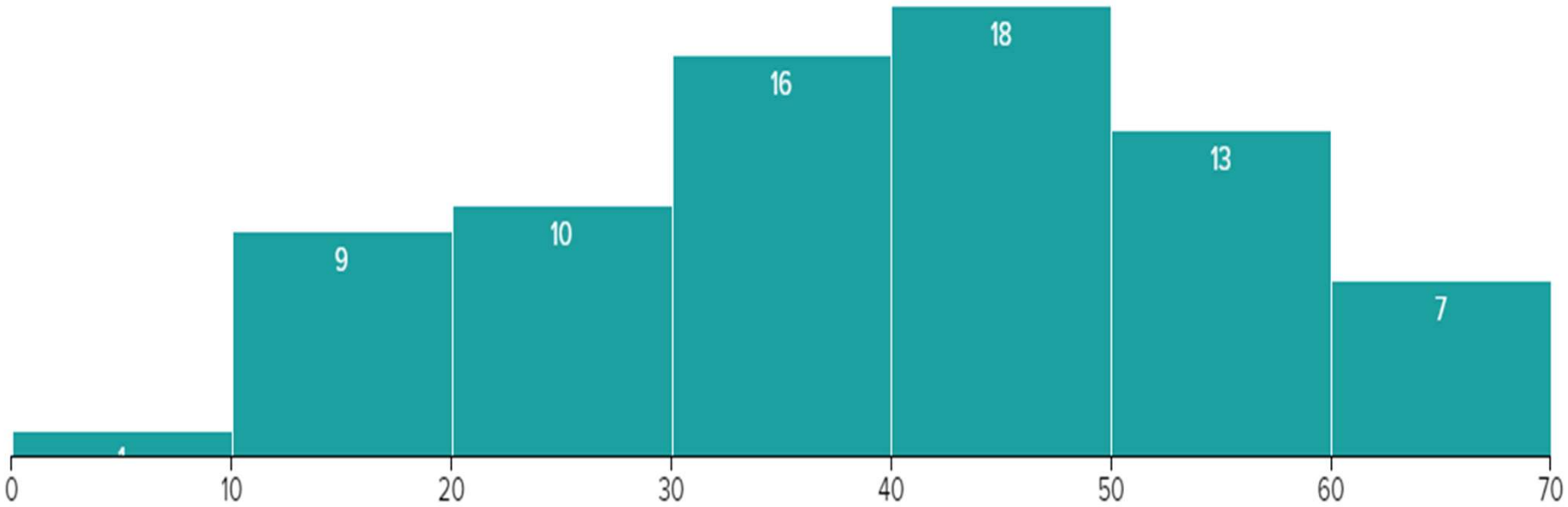
18-447 Lecture 13: Bus, Protocol, and I/O

James C. Hoe
Department of ECE
Carnegie Mellon University

Midterm 1 Class Distribution



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
8.0	40.25	70.0	39.85	15.12

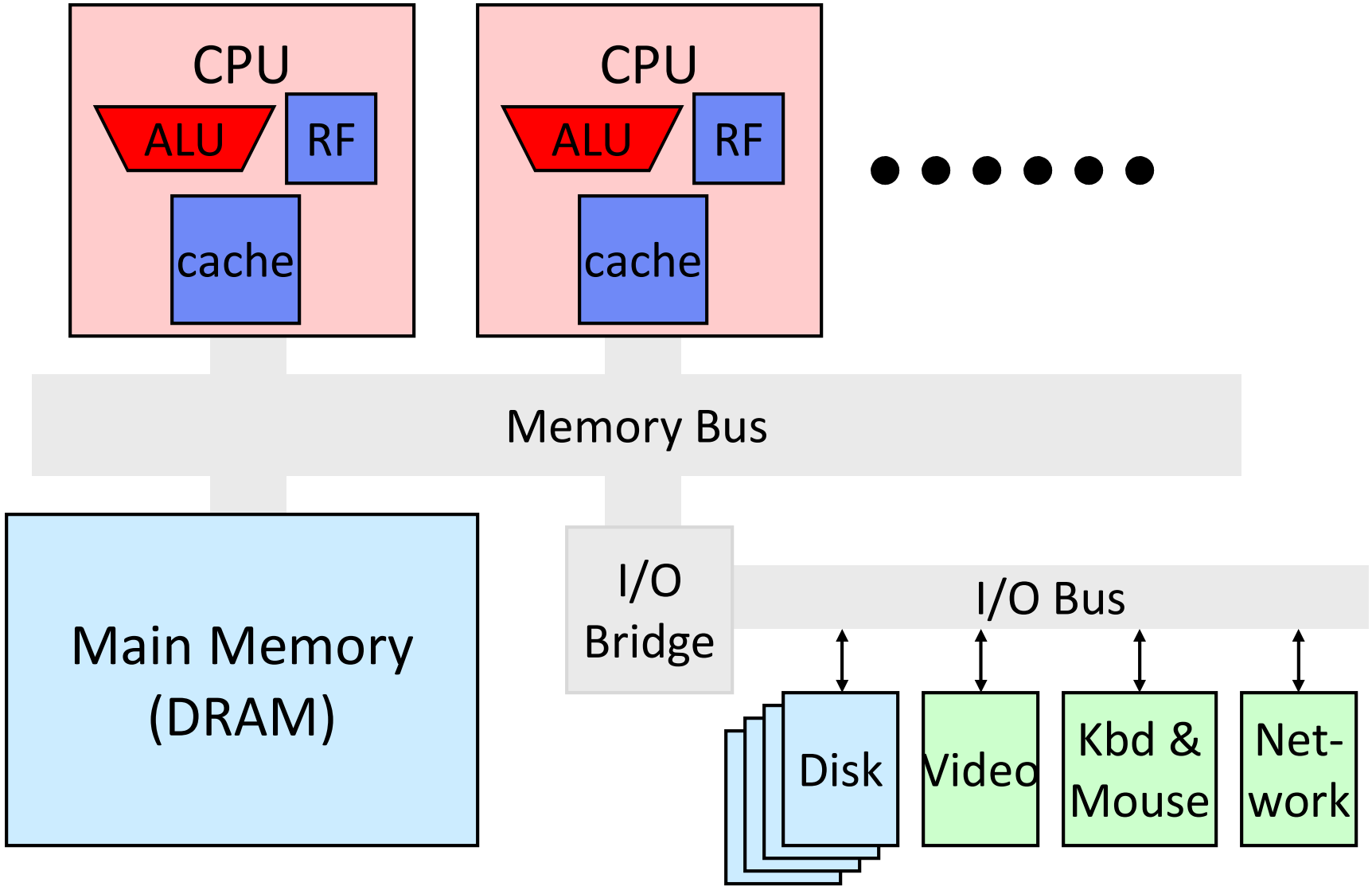
Midterm 1 Summary Statistics

	1:AUIPC	2:pipeln	3:IPC	4:hzrd	5:BP	6:ucode	7:assmb	total
possible	4	6	12	12	12	10	14	70
average	**2.0**	**1.9**	**5.3**	**6.5**	**10.2**	**5.4**	**8.6**	**39.9**
stdev	**1.9**	**2.2**	**4.0**	**4.0**	**2.4**	**3.3**	**4.8**	**15.1**
max	**4.0**	**6.0**	**12.0**	**12.0**	**12.0**	**10.0**	**14.0**	**70.0**
median	**3.0**	**0.8**	**4.0**	**7.3**	**12.0**	**6.0**	**10.0**	**40.3**
min	**0.0**	**0.0**	**0.0**	**0.0**	**2.0**	**0.0**	**0.0**	**8.0**

Housekeeping

- Your goal today
  - see how components in a system hang together
  - see how decoupled units interoperate by "protocol"
- Notices
  - Lab 3, due week 10 (Handout #12 on Canvas)
  - HW 4, due 3/21
  - Midterm 2, Wed, 4/6, covers up to Lec 18
- Readings
  - start reading P&H Ch5 . . .
  - http://en.wikipedia.org/wiki/Conventional_PCI
  - http://en.wikipedia.org/wiki/PCI_Express

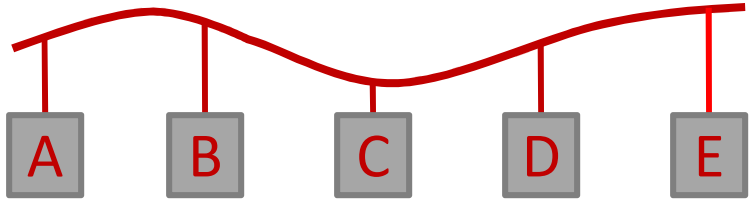
Classic View of Computer System



"Broadcast" or "True" Bus

- Common wires connecting multiple devices
  - multiple drivers and multiple receivers, but one driver at a time broadcast
  - time-multiplexed shared usage by "transactions"

As opposed to point-to-point



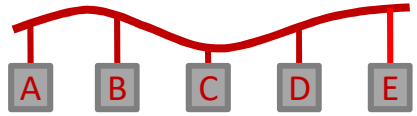
- Good idea if
  - high board-level wire cost
  - low individual bandwidth requirement
  - low aggregate bandwidth requirement
- Standardized connections and protocol for system expansion

Bus Transaction

- Device types
  - **initiators:** devices that can initiate transactions
  - **targets:** devices that only respond
  - **arbiter:** a special device that manages sharing
- Memory-like paradigm
  - "address", "data", "reading vs. writing"
  - initiator issues read/write **request** to an address
  - each target assigned an address range to **respond** for, by returning or accepting data

To start, visualize processor as initiator and memory as target device; trxn's stem from program's LW/SW

Bus Transaction Phases



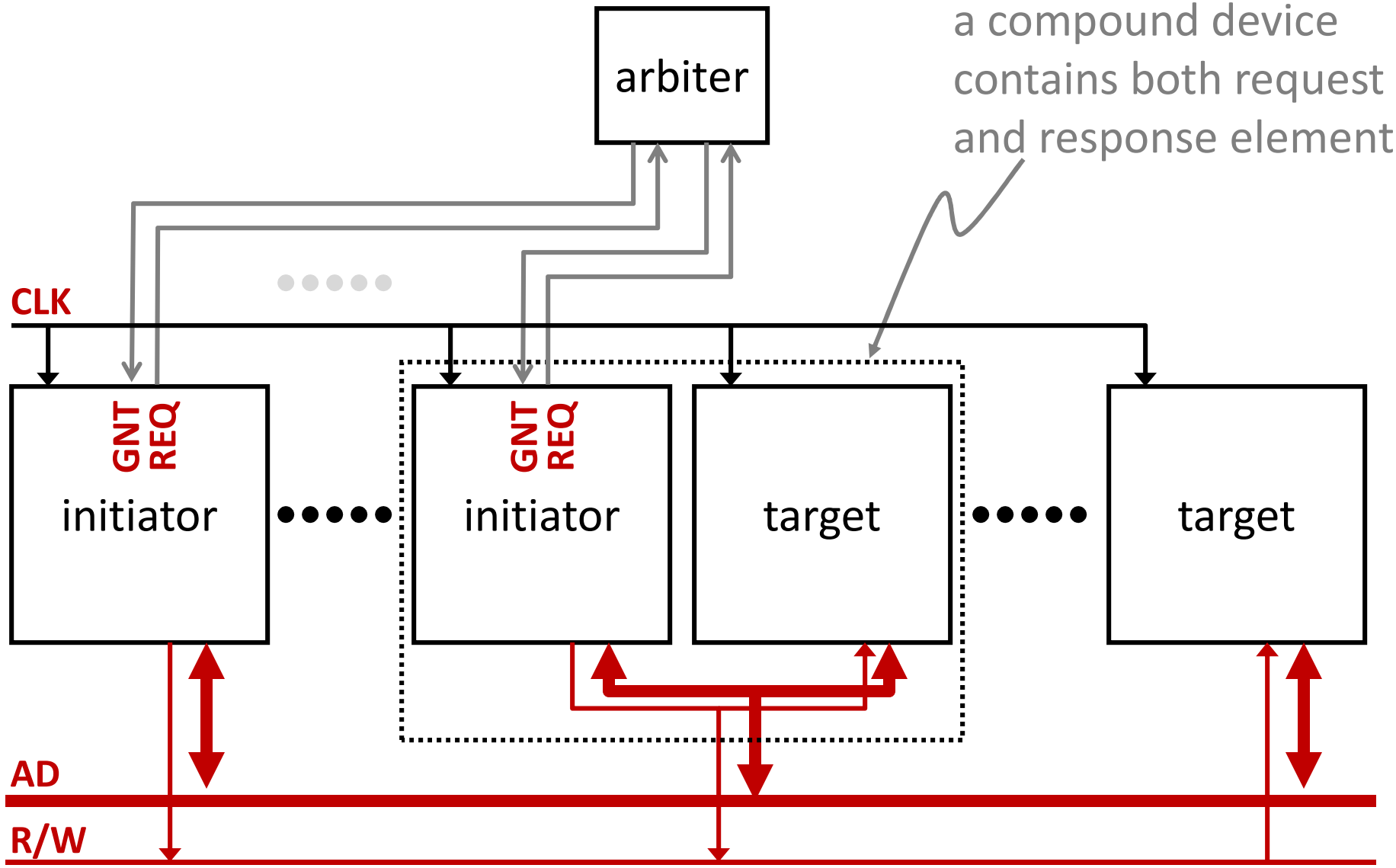
1. Arbitration Phase
   - 1 or more initiators **request** ownership
   - arbiter **grants** ownership to 1 initiator
2. Address Phase
   - initiator drives **address** for all to see
   - 1 target **claims** transaction
3. Data Phase
   - initiator (or target) drives write (or read) **data** for all to see
4. Termination Phase:
   - initiator **terminates** bus ownership

"Bus Protocol" defines exact signals and rules of conduct

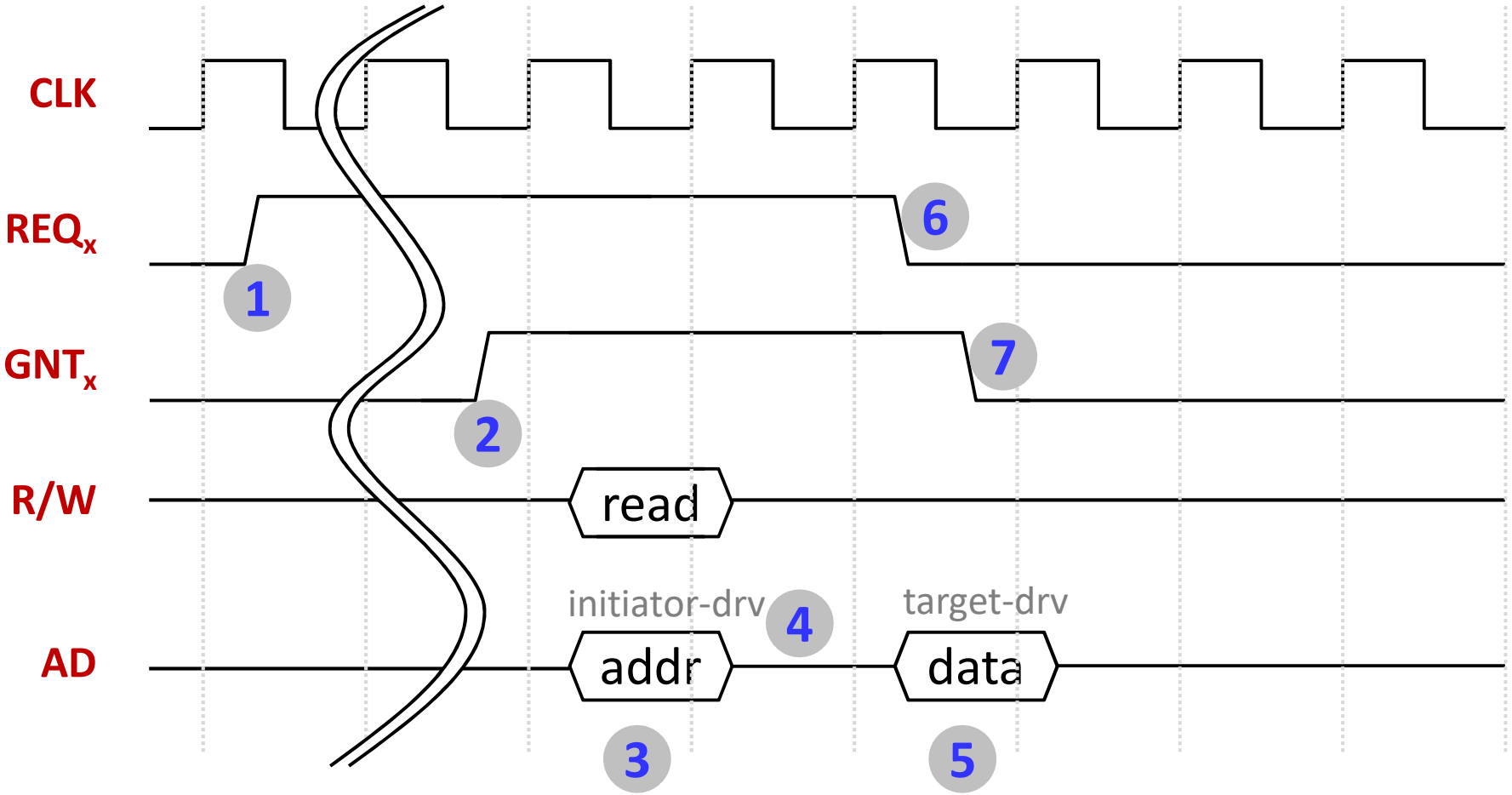
Basic Bus Signals

- **CLK**: all devices synchronized by a common clock
- Per-initiator point-to-point signals to/from arbiter
  - **REQ** (initiator→arbiter): assert to request ownership; de-assert to signal end of transaction
  - **GNT** (arbiter→initiator): ownership is granted
- "Broadcast" signals shared by all devices
  - **AD[]** (address/data bus, bi-directional): initiator drives address during address phase, initiator or target drives data during data phase
  - **R/W** (bi-directional): commands, e.g., read vs. write

Bus Configuration

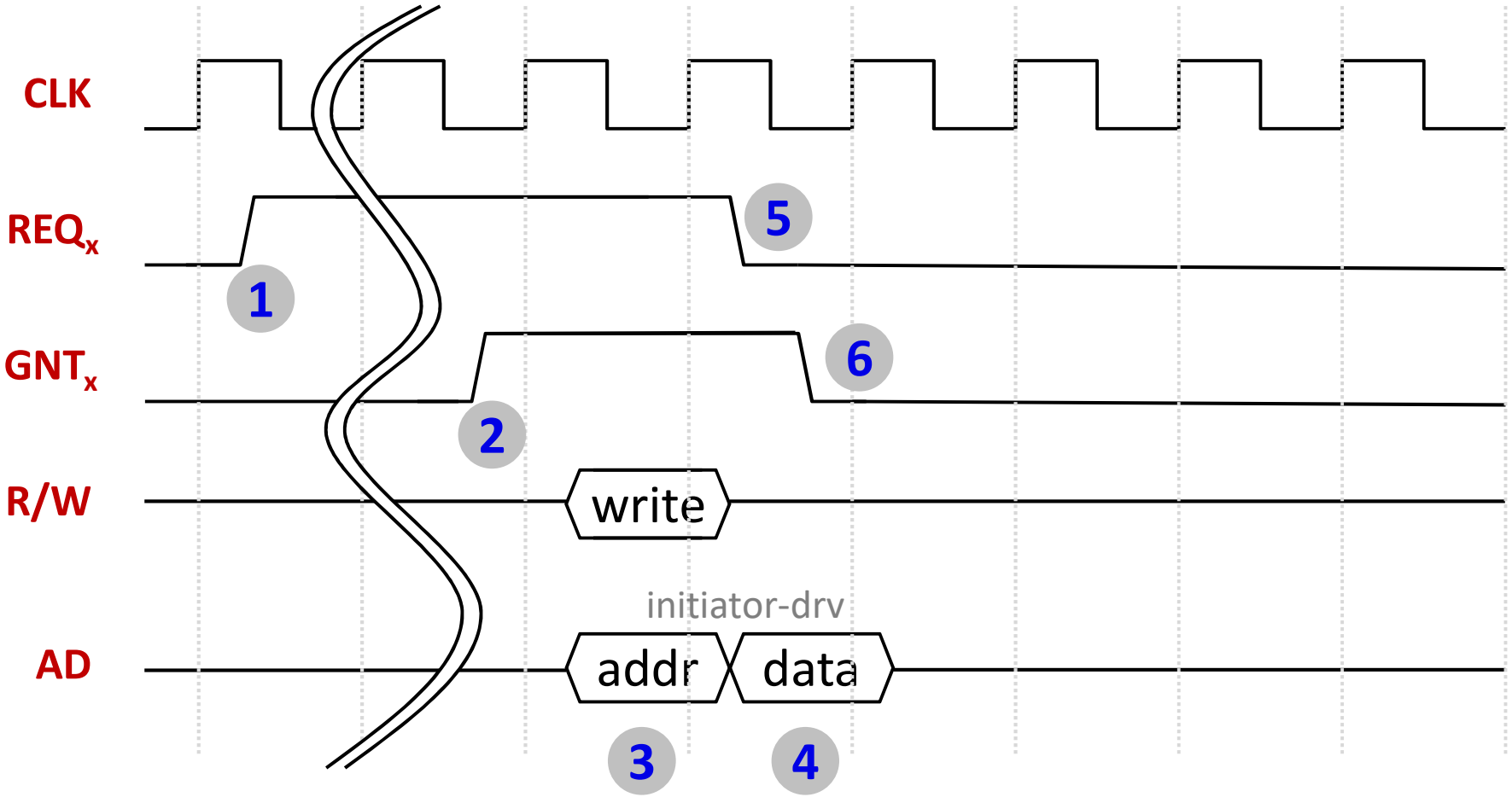


Simple Read Transaction



1. initiator$_x$ requests bus
2. arbiter grants bus
3. initiator$_x$ drives address/command, to be sampled on clock-edge
4. bus-turnaround cycle
5. target drives data
6. initiator$_x$ signals final cycle
7. arbiter acknowledges

Simple Write Transaction

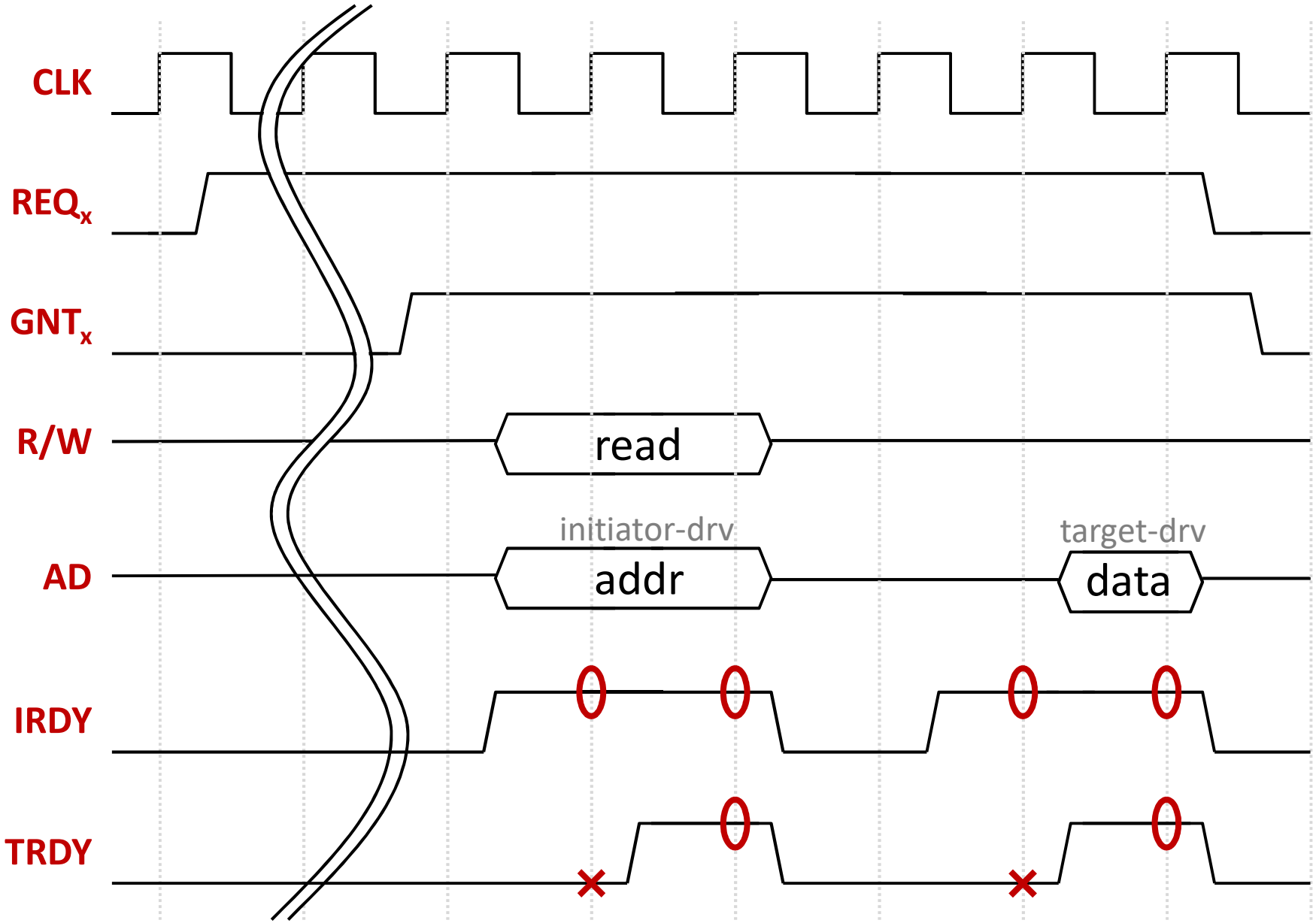


1. initiator$_x$ requests bus
2. arbiter grants bus
3. initiator$_x$ drives address/command, to be sampled on clock-edge
4. initiator$_x$ drives data
5. initiator$_x$ signals final cycle
6. arbiter acknowledges

Asynchronous Protocols

- "Synchronous" bus protocol has fixed timing
  - targets must react fast enough
  - bad when mixing slow and fast targets (e.g., on I/O expansion bus)
- Asynchronous handshaking
  - **REQ/GNT** is an example of asynchronous handshake
  - elastic amount of time to respond
- Asynchronous bus protocols
  - add **IRDY** and **TRDY** for initiator and target
  - **AD** valid only when **IRDY**&&**TRDY**
    - receiver pays attention only if driver is ready
    - driver repeats value until receiver is ready
    - both driver and receiver can delay arbitrarily

Async Read Transaction



Bus Performance: Latency

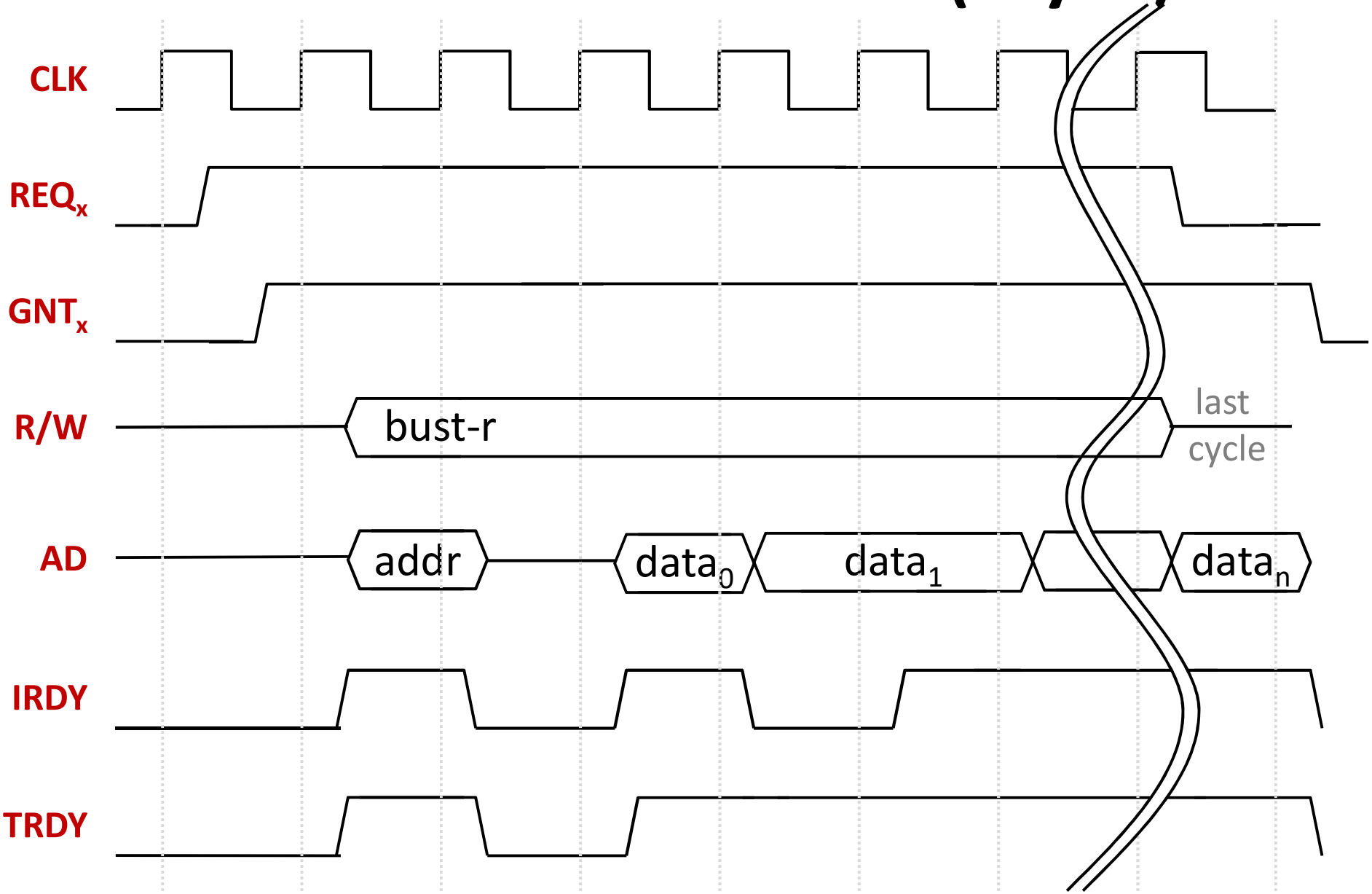
- Request/Grant latency depends on
  - degree of bus contention
  - arbitration strategy under contention
    - statically prioritized by expansion slots
    - FIFO, round-robin (and other so-called fair arbitrations)
- Transaction latency depends
  - target reaction time
  - transfer size

Keep in mind, actual latency felt by program LW/SW much longer than raw bus latency

Bus Performance: Bandwidth

- Peak Bandwidth
  - assume **w**-byte **AD** bus at frequency **f**
  - BW_{peak} = **w**·**f** "guaranteed not to exceed"
- Effective BW deducts for overhead cycles
  - request and grant phases
  - address and claim phases
  - termination phase
- Best if overhead *amortized* over many data cycles
  - burst access to successive consecutive addresses
  - fixed-sized burst on synchronous protocols
  - variable-sized burst on asynchronous protocols

Burst Read Transaction (async)



Effective Bandwidth Quantified

- Effective BW is fxn of number of "data beats"
  - **w** = bus width in bytes;
  - **t** = bus cycle time, 1/**f**
  - **v** = # cycles in overhead; **n** = # data cycles
- $BW_{effective} = \mathbf{n}\cdot\mathbf{w} \;/\; (\mathbf{v}\cdot\mathbf{t} + \mathbf{n}\cdot\mathbf{t})$
  - if (**n**=1)<<**v**, $BW_{effective} \approx \mathbf{w}/(\mathbf{v}\cdot\mathbf{t})$
  - if **n** >> **v**, $BW_{effective} \approx BW_{peak} = \mathbf{w}/\mathbf{t}$
- E.g., **f**=33MHz, **w**=4, BW_{peak} = 133MB/s (PCI 1.0)
  - simple read, 3 AD cycles, **v**=2, **n**=1

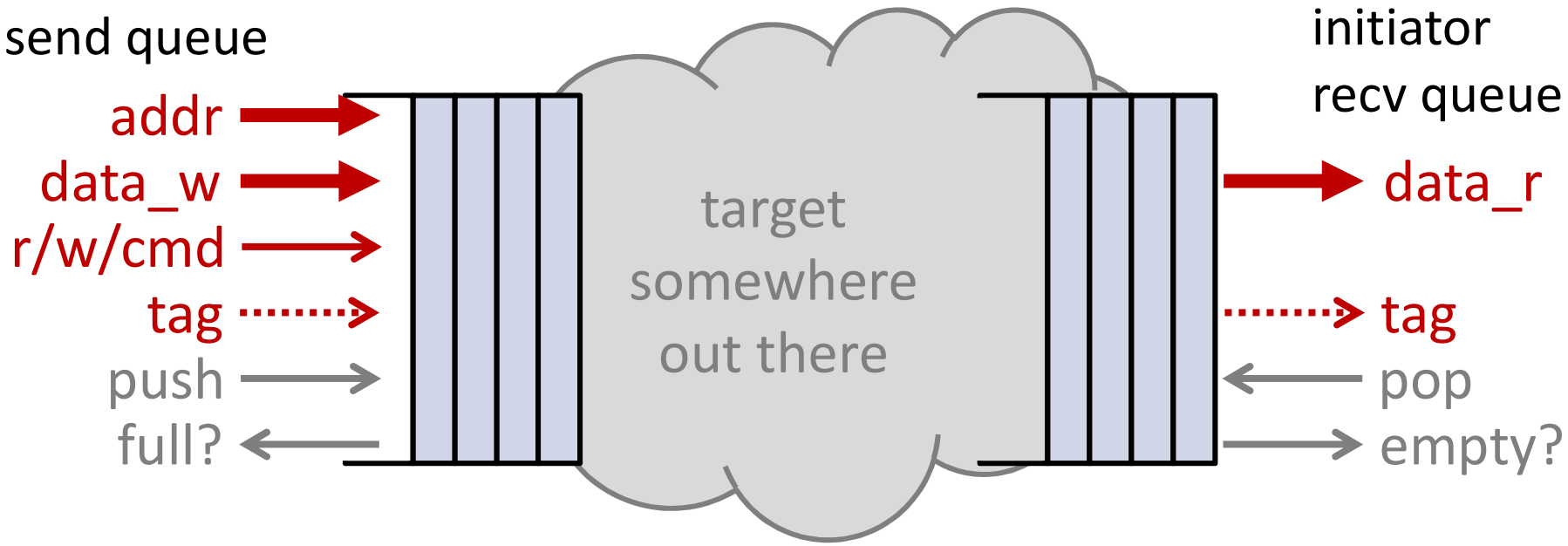
    $BW_{effective}$ = 44 MB/s
  - burst read

    $BW_{effective,n=2}$ = 66 MB/s; $BW_{effective,n=16}$ = 118 MB/s

Advanced Bus Architectures

- Pipelined bus
  - separate address and data bus
  - overlap request/address/data phases of 3 trxn's
- Out-of-order (aka. split-phase) bus
  - separate arbitration for address and data bus
  - address-bus trxn is assigned an unique **tag**;
  - target arbitrates for data bus when ready; use **tag** to identify initiator; data phase out-of-order!
- Switched data bus
  - split-phase bus with true address bus
  - but crossbar for data bus to achieve high BW
- Point-to-point "bus"

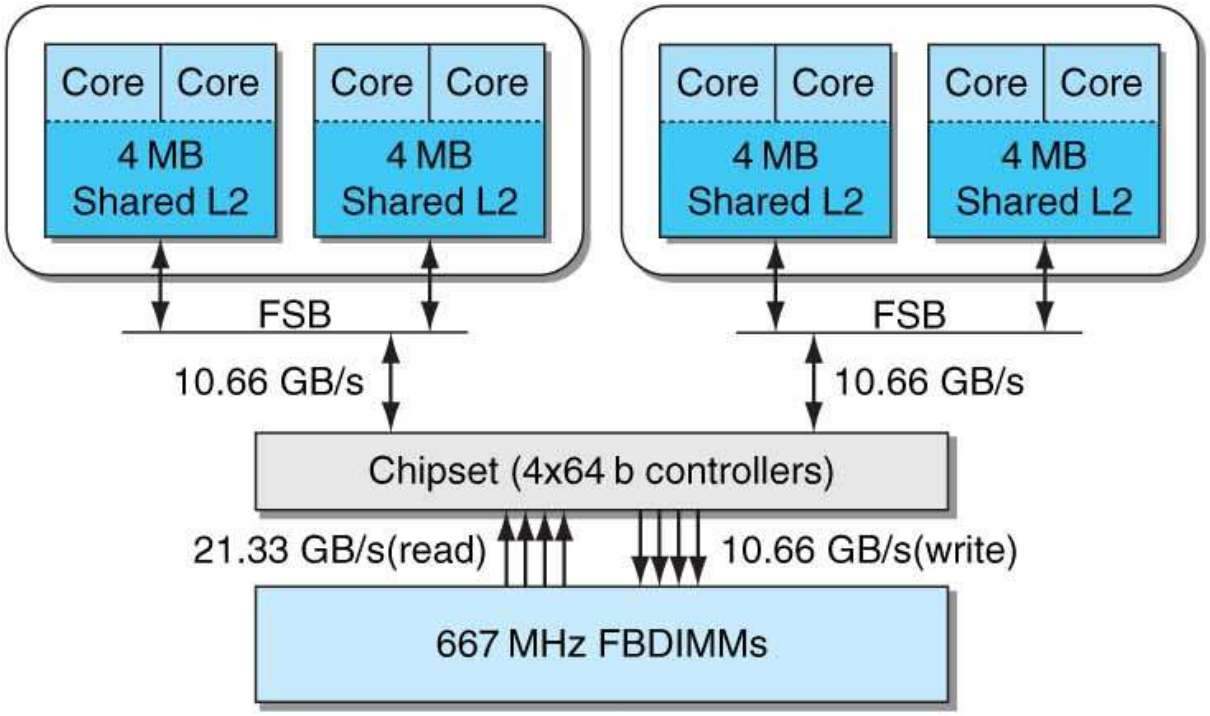
Point-to-Point "Bus"



- Same memory-like read and write transactions, but . .. .
- Split-phase transactions via message passing
  - initiator sends read/write request message
  - request routed to target based on "bus" address
  - target sends data/ack message
  - reply message routed back to initiator

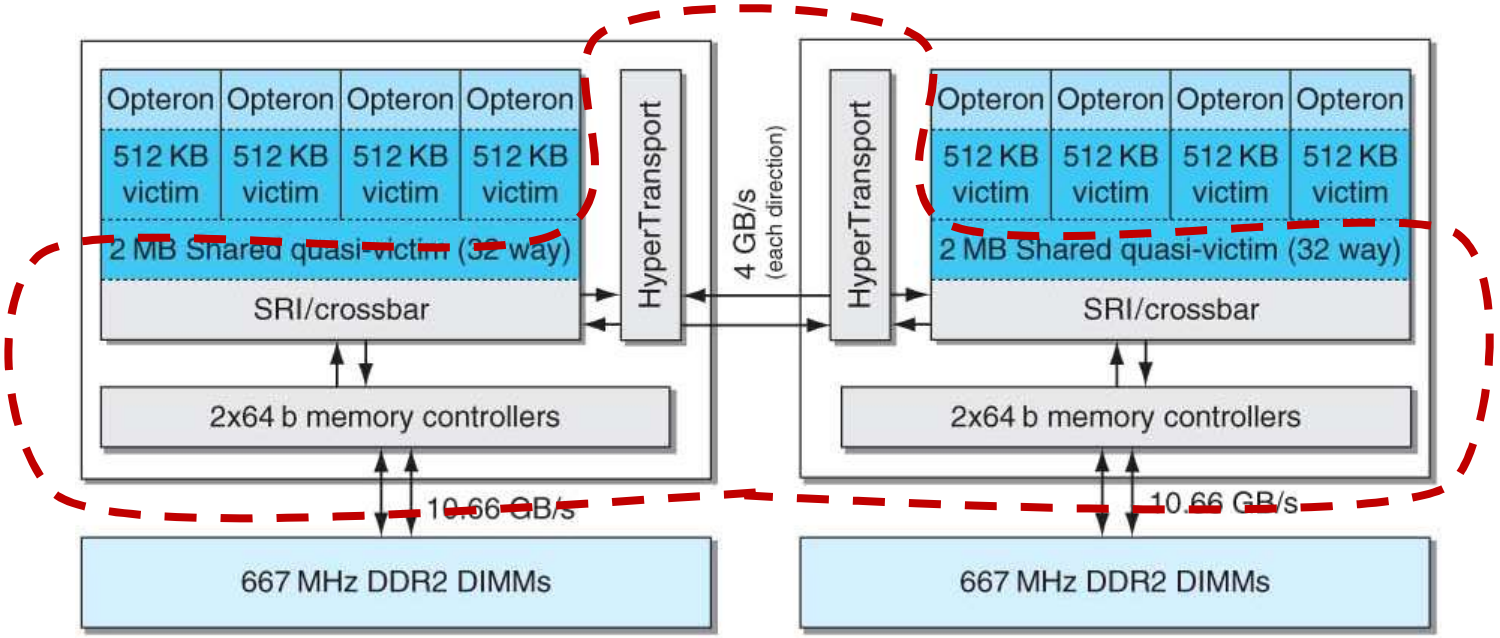
No arbitration or claim

Intel Xeon e5345 (Clovertown)



Busses no longer monolithic

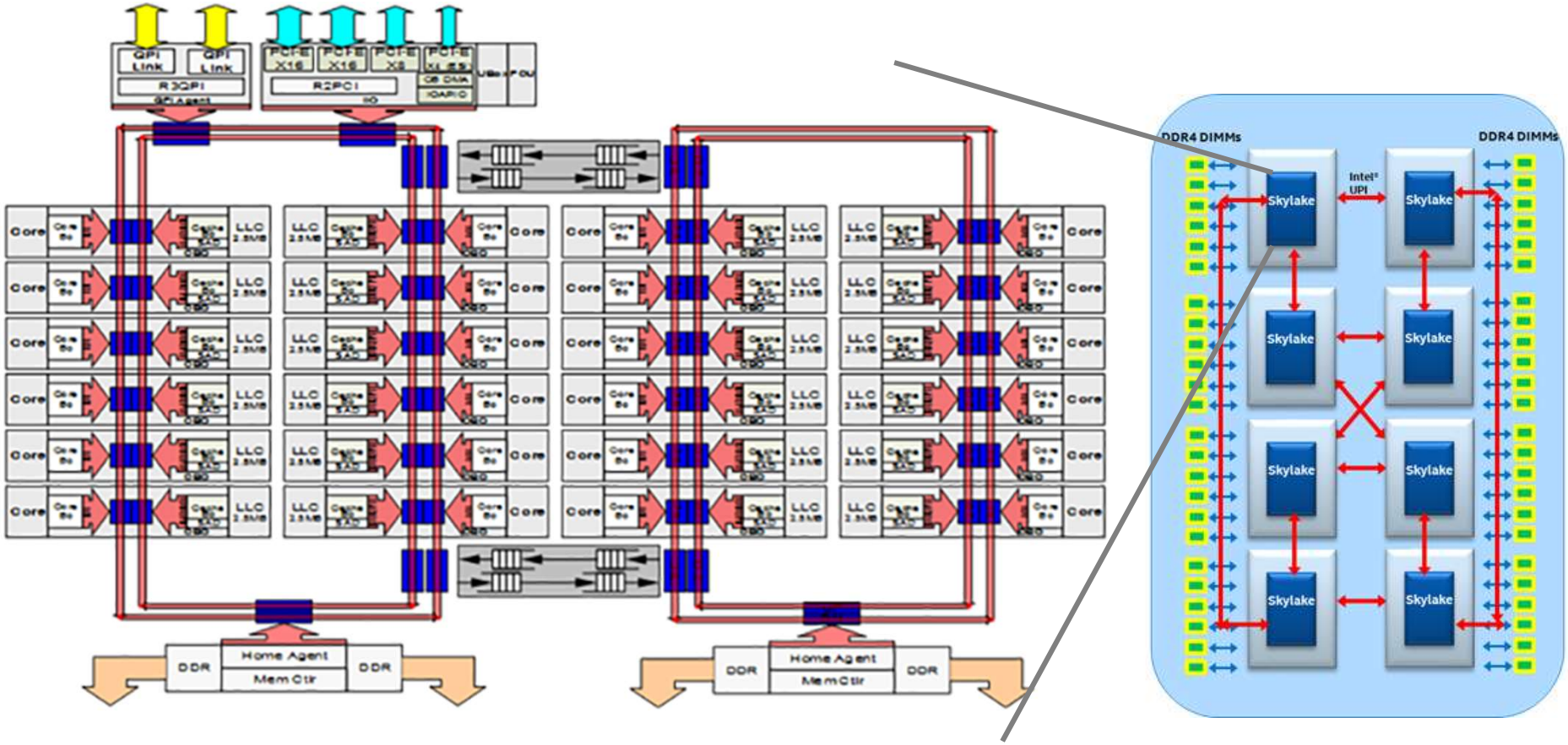
AMD Opteron X4 2356 (Barcelona)



[Figure from P&H CO&D, COPYRIGHT 2009 Elsevier. ALL RIGHTS RESERVED.]

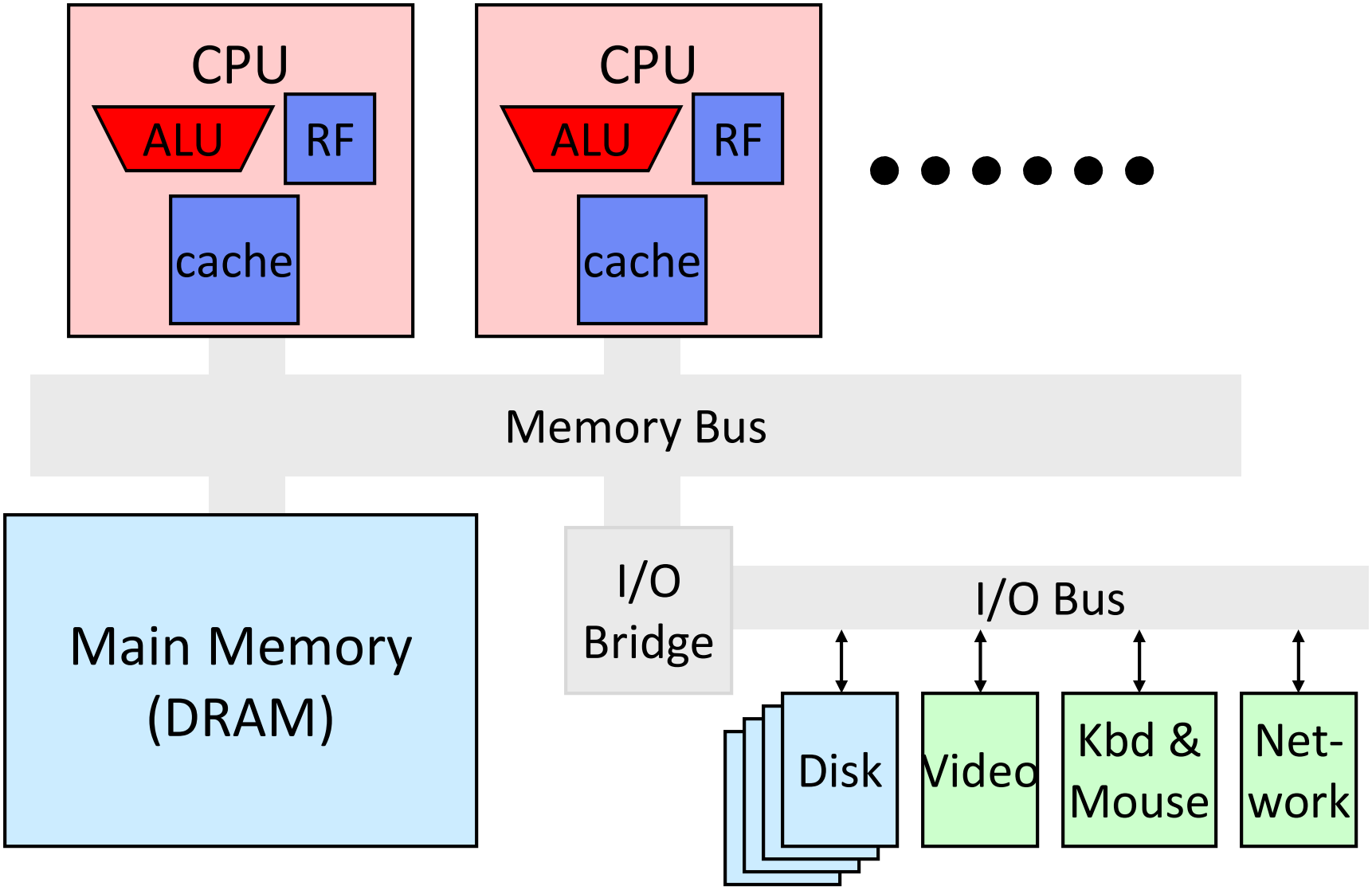
There is no arbitrated broadcast "bus" anywhere but interactions still based on initiator/target and read/write transactions to addresses

Intel "Uncore" Architecture



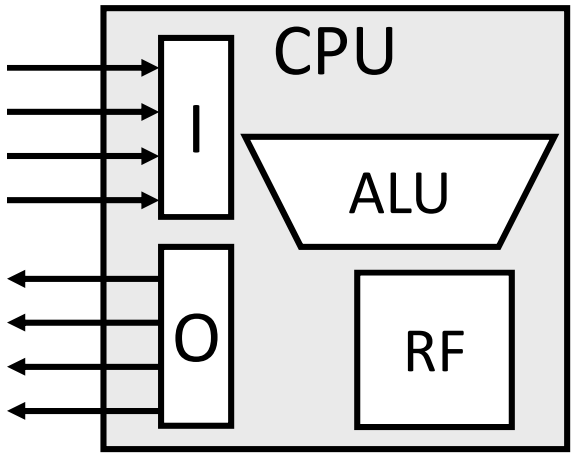
[https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview]

How do processors talk to I/O?



The Easiest: I/O Port Registers

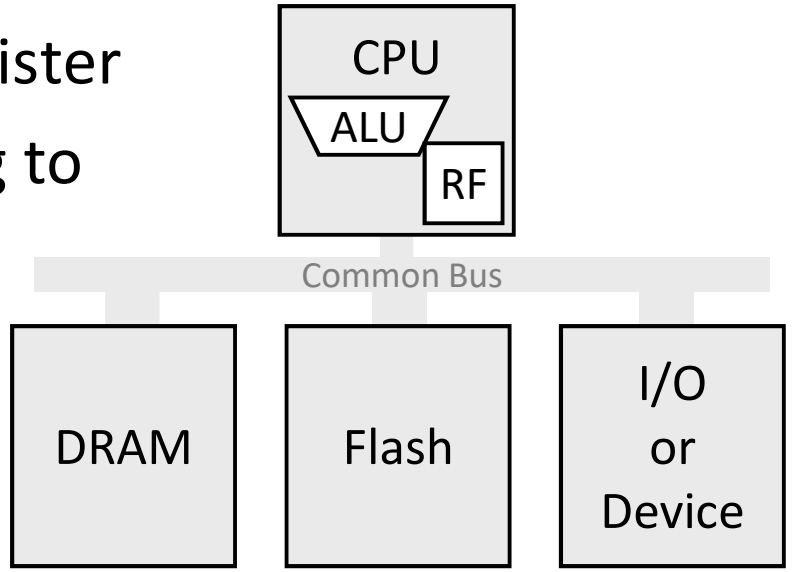
- I/O registers as ISA programmer-visible state
  - **output:** write value appear on output pins
  - **input:** reading returns values on input pins
- Common scheme on microcontrollers
  - easy to use, low latency
  - can be specialized for application
- Not general
  - predetermined number of I/O
  - specialized for whose application?



Is there a general I/O scheme?

Generalized Memory-Mapped I/O

- Memory load/store is a kind of I/O
  - address identifies a specific memory location
  - read/write convey data from/to memory
- "Map" unused memory addresses (e.g., the high ones) to registers of external devices
  - LW from "mmap" address means moving data from the corresponding register
  - similarly, SW means moving to

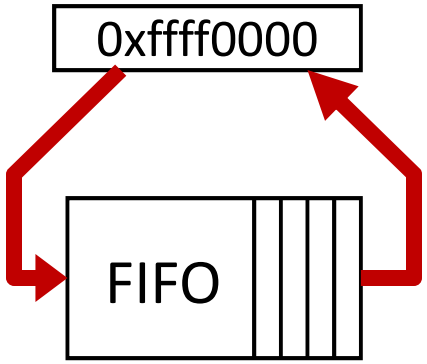


memory and I/O devices respond to their assigned address ranges

Idempotency and Side-effects

- Loading from real memory location M[A] should return most recent value stored to M[A]
  - ⇒ writing M[A] once is the same as writing M[A] with same value multiple times in a row
  - ⇒ reading M[A] multiple times returns same value

  This is why memory caching works!!
- LW/SW to mmap locations can have side-effects
  - reading/writing mmap location can imply commands and other state changes
  - e.g., a mmap device that is a FIFO
    - SW to 0xffff0000 pushes value
    - LW from 0xffff0000 returns popped value



What happens if 0xffff0000 is cached?

Direct Memory Access

- mmap I/O is slow and consumes processor cycles

How slow?

- Why not let I/O devices access memory directly
- Processor program DMA device by mmio
  - e.g., "read (or write) 1024 KBytes starting from location 0x54100"
  - DMA device read/write memory directly
  - only makes sense for moving large data blocks

Does DMA device see cached values?

- How does the processor know when a DMA transfer is finished?

Use #1: Interrupts

- How to handle rare events with unpredictable arrival time and must be acted upon quickly?

  E.g., keystroke, in-bound network, disk I/O

- **Option 1:** write every program with periodic calls to a service routine
  - polling frequency affects worst-case response time
  - expensive for rare events needing fast response

  What if a programmer forgets to do it?

- **Option 2:** normal programs blissfully unaware
  - event triggers an interrupt on-demand
  - forcefully and transparently transfer control to the service routine and back

Recall

Polling I/O

- Option 1 but done by kernel in loop or timer interrupt
- Consider a keyboard with 2 read-only registers
  - **READY**: returns true if a new character is available
  - **DATA**: returns next character in kbd buffer;
    and **resets** **READY** if no more characters
- Polling-based service routine


```
_checkkbd: LW  r16 _READY
           BEQ r16 r0 _end
           LW  r3 _DATA
           JAL _handle_keystroke
           J _checkkbd
_end:      JR  r31
```

Interrupt-Driven I/O

- How frequently to poll?
  - how fast can you type?
  - how fast can you see what you type?

  Polling is expensive when above very different
- Give keyboard (or an I/O class) an interrupt line
  - keyboard raise interrupt on new keystroke
  - interrupt handler triage and call **`_checkkbd`**
- Interrupt best suited for infrequent/irregular events with tight service latency requirement

  Polling okay for keyboard but not network, what about DMA?

Polling vs Interrupt

- Consider for an I/O device
  - average interarrival time t_a between events
  - max reaction time t_r to service event
- Must poll faster than 1/t_r to keep deadline
  - $t_a << t_r$: could poll upto 1/t_a and be productive
  - $t_a >> t_r$: polling at 1/t_r very wasteful
    - t_a regular or predictable: poll when expected
    - unpredictable: good use case for interrupt
  - except stay with polling if
    - processor has nothing else to do anyways
    - t_r so large polling overhead negligible
    - t_r so tight interrupt handler already late at start

Which I/O Mechanism to use?

- First, depends on what you are doing
- Second, limited by what is available
- Performance considerations
  - I/O Bandwidth = transfer size / transfer time
  - Transfer time = overhead + (transfer size / BWraw)
    - DMA: high bandwidth but large setup overhead
    - mmio: low bandwidth but no overhead
- Processor considerations
  - what fraction of processor time lost to I/O?
  - does processor have other user tasks to do?
  - how long can I/O wait?