Interfacing

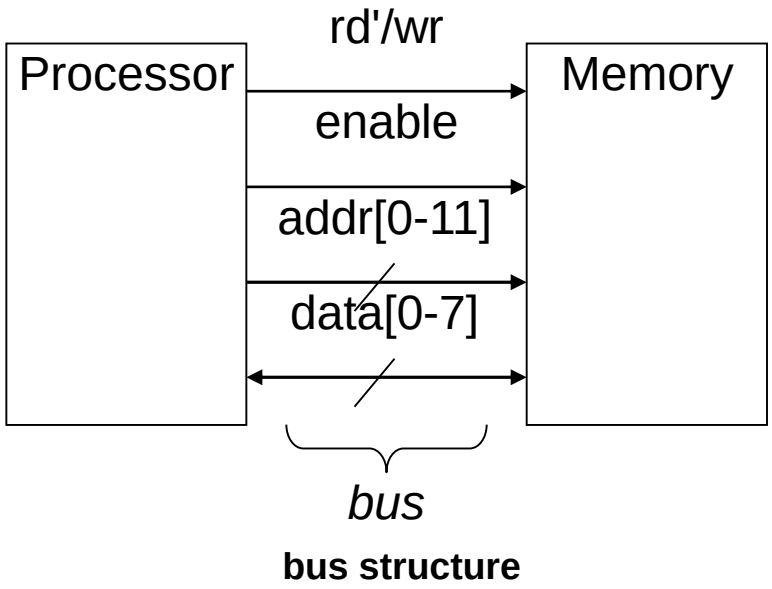
Shao-Yi Chien

Outline

- Interfacing basics
- uP interfacing: I/O Addressing
- uP interfacing: Interrupts
- uP interfacing: Direct memory access
- Arbitration
- Hierarchical buses

A Simple Bus

- Wires:
  - Uni-directional or bi-directional
  - One line may represent multiple wires
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication

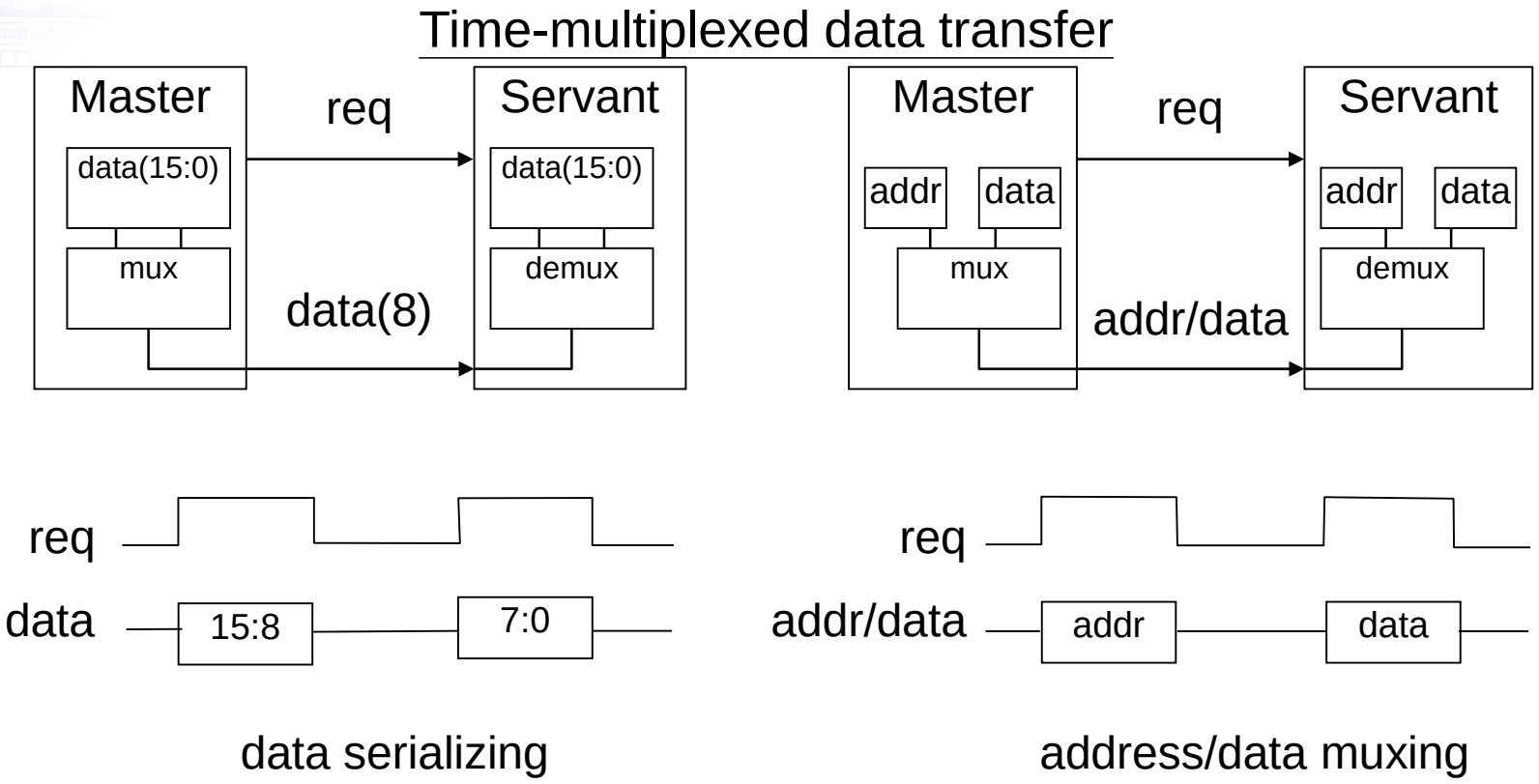


bus structure

Basic Protocol Concepts

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Addresses: special kind of data
  - Specifies a location in memory, a peripheral, or a register within a peripheral
- Time multiplexing
  - Share a single set of wires for multiple pieces of data
  - Saves wires at expense of time

Example of Time Multiplexing

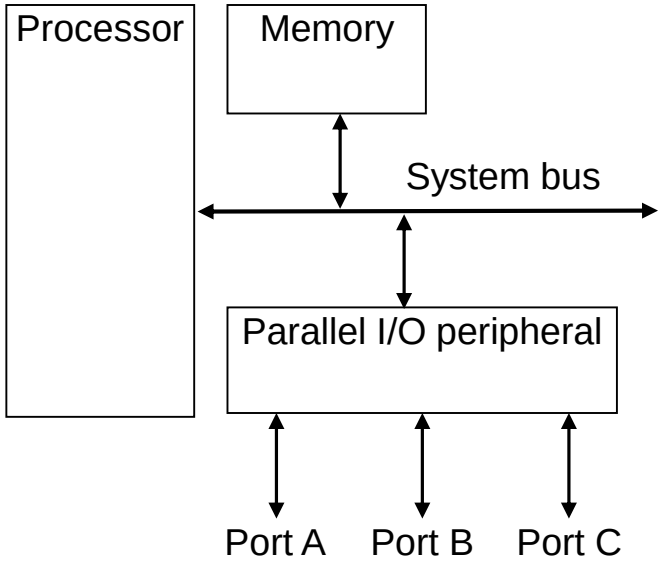


uP Interfacing: I/O Addressing

- A microprocessor communicates with other devices using some of its pins
  - **Port-based I/O (parallel I/O)**
    - Processor has one or more N-bit ports
    - Processor's software reads and writes a port **just like a register**
    - Ex: P0 = 0xFF; v = P1.2; -- P0 and P1 are 8-bit ports (can be accessed bit by bit)
  - **Bus-based I/O**
    - Processor has address, data and control ports that form a single bus
    - Communication protocol is built into the processor
    - A single instruction carries out the read or write protocol on the bus

Compromises/Extensions

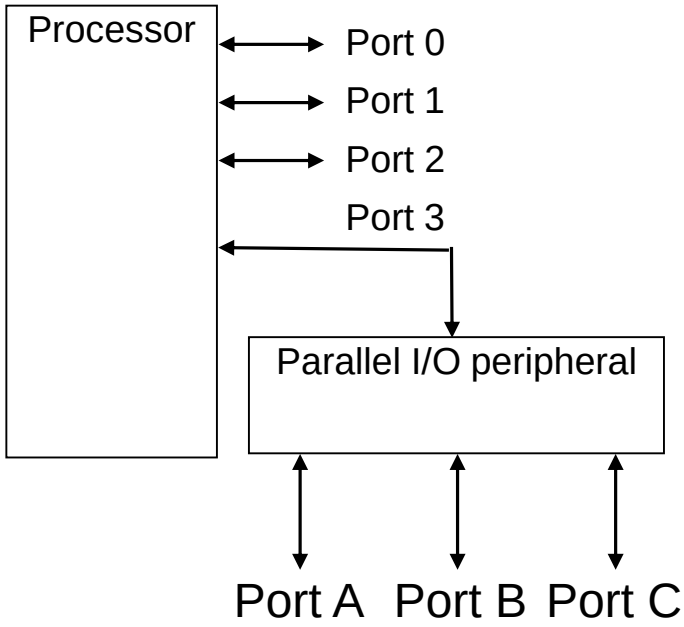
- Parallel I/O peripheral
  - When processor only supports bus-based I/O but parallel I/O is needed
  - Each port on peripheral connected to a register within peripheral that is read/written by the processor



Adding parallel I/O to a bus-based I/O processor

Compromises/Extensions

- Extended parallel I/O
  - When processor supports port-based I/O but more ports are needed
  - One or more processor ports interface with parallel I/O peripheral extending total number of ports available for I/O



Extended parallel I/O

Types of Bus-Based I/O

- Processor talks to both memory and peripherals using the same bus – two ways to talk to peripherals
  - **Memory-mapped I/O**
    - Peripheral registers occupy addresses in the same address space **as memory**
    - e.g., Bus has 16-bit address
      - lower 32K addresses may correspond to memory
      - upper 32k addresses may correspond to peripherals
  - **Standard I/O (I/O-mapped I/O)**
    - **Additional pin** (*M/IO*) on bus indicates **whether a memory or peripheral access**
    - e.g., Bus has 16-bit address
      - all 64K addresses correspond to memory when *M/IO* set to 0
      - all 64K addresses correspond to peripherals when *M/IO* set to 1

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

- Memory-mapped I/O
  - Requires no special instructions
    - Assembly instructions involving memory like MOV and ADD work with peripherals as well
    - Standard I/O requires special instructions (e.g., IN, OUT) to move data between peripheral registers and memory
- Standard I/O
  - No loss of memory addresses to peripherals
  - Simpler address decoding logic in peripherals possible
    - When number of peripherals is much smaller than address space then high-order address bits can be ignored → smaller and/or faster comparators

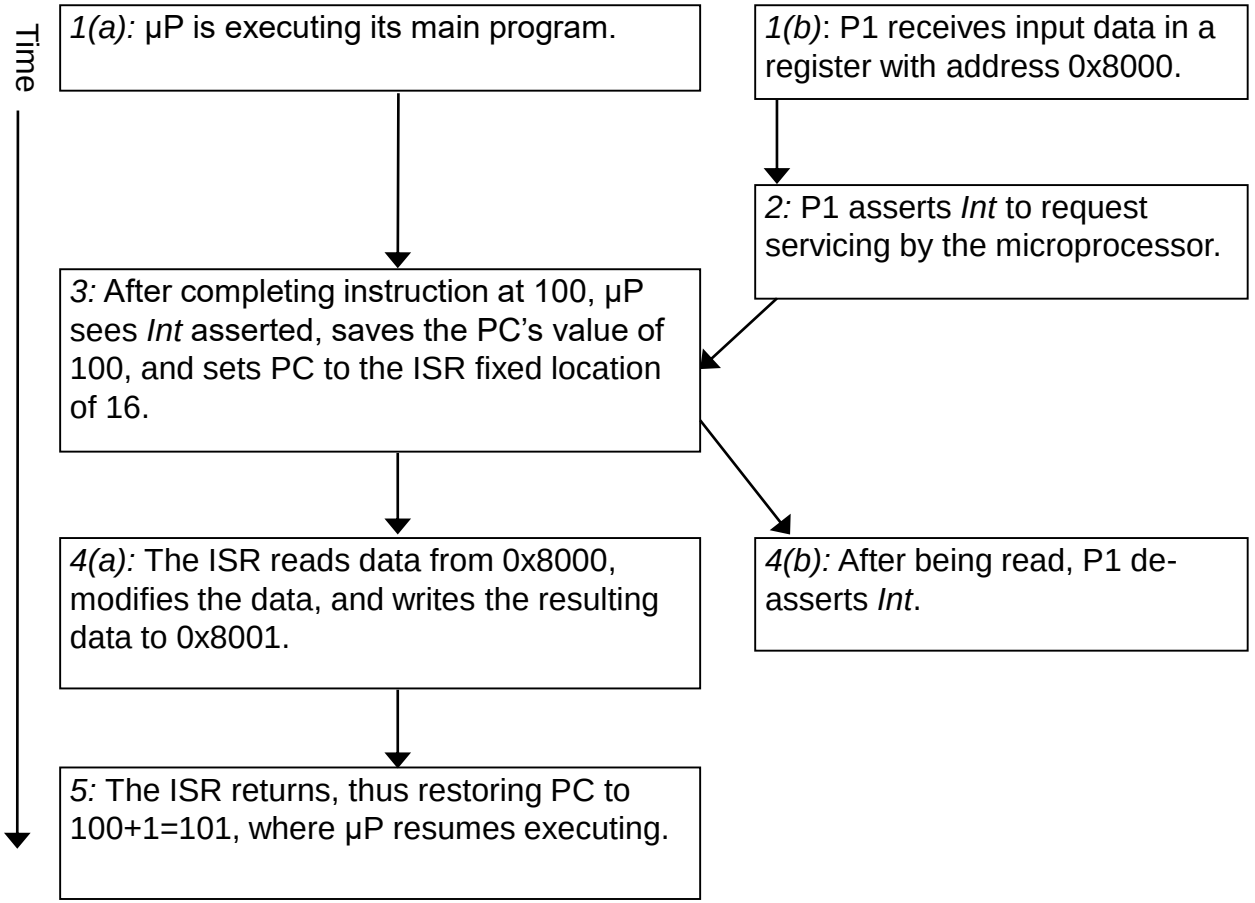
uP Interfacing: Interrupts

- Suppose a peripheral intermittently receives data, which must be serviced by the processor
  - The processor can ***poll*** the peripheral regularly to see if data has arrived – wasteful
  - The peripheral can ***interrupt*** the processor when it has data
- Requires an extra pin or pins: Int
  - If Int is 1, processor suspends current program, jumps to an **Interrupt Service Routine**, or **ISR**
  - Known as interrupt-driven I/O
  - Essentially, "polling" of the interrupt pin is built-into the hardware, so no extra time!

uP interfacing: Interrupts

- What is the address (interrupt address vector) of the ISR?
  - **Fixed interrupt**
    - Address built into microprocessor, cannot be changed
    - Either ISR stored at address or a jump to actual ISR stored if not enough bytes available
  - **Vectored interrupt**
    - Peripheral must provide the address
    - Common when microprocessor has multiple peripherals connected by a system bus
  - **Compromise: interrupt address table**

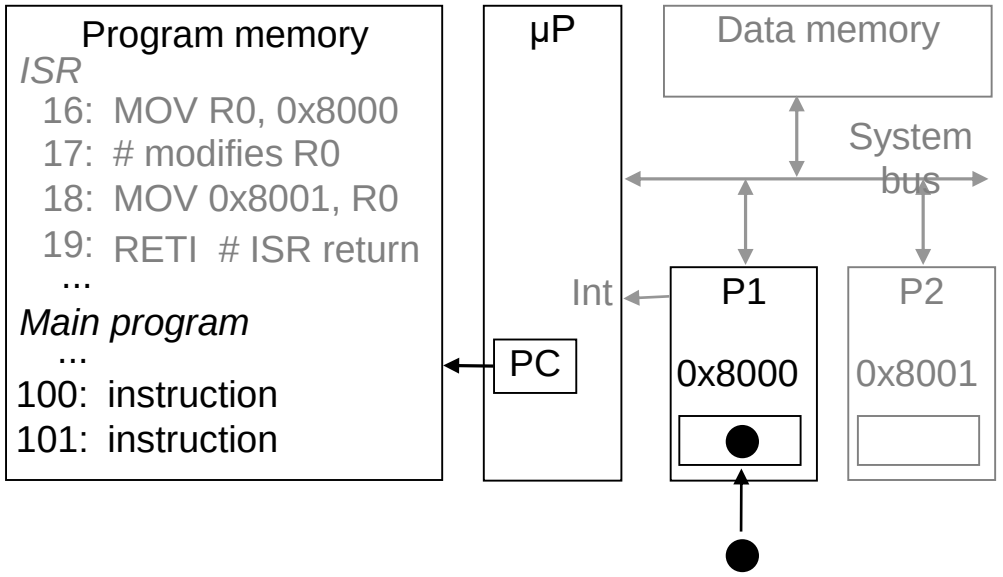
Interrupt-Driven I/O using Fixed ISR Location



Interrupt-Driven I/O using Fixed ISR Location

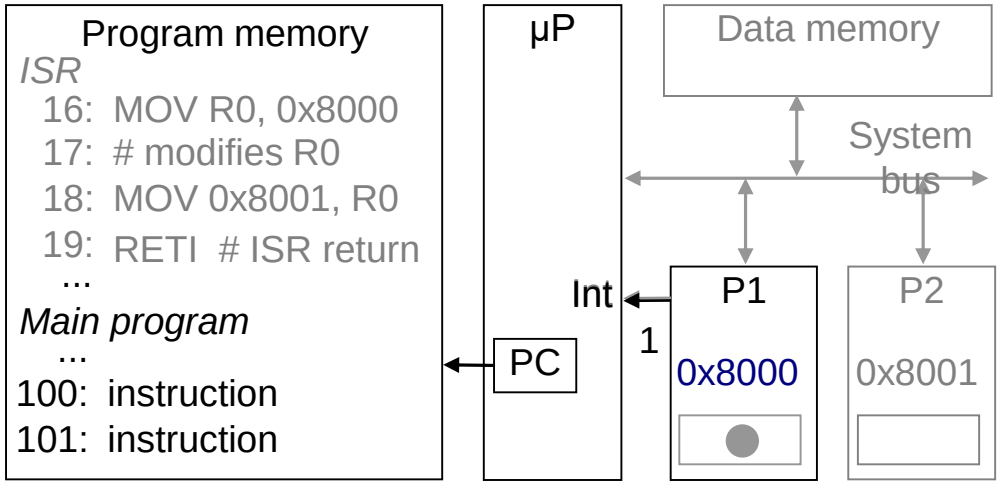
1(a): μP is executing its main program

1(b): P1 receives input data in a register with address 0x8000.



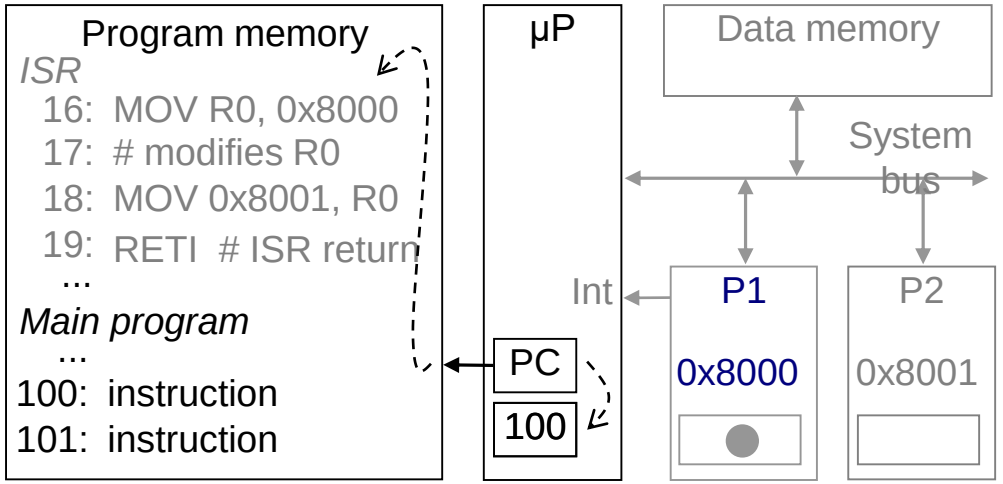
Interrupt-Driven I/O using Fixed ISR Location

2: P1 asserts *Int* to request servicing by the microprocessor



Interrupt-Driven I/O using Fixed ISR Location

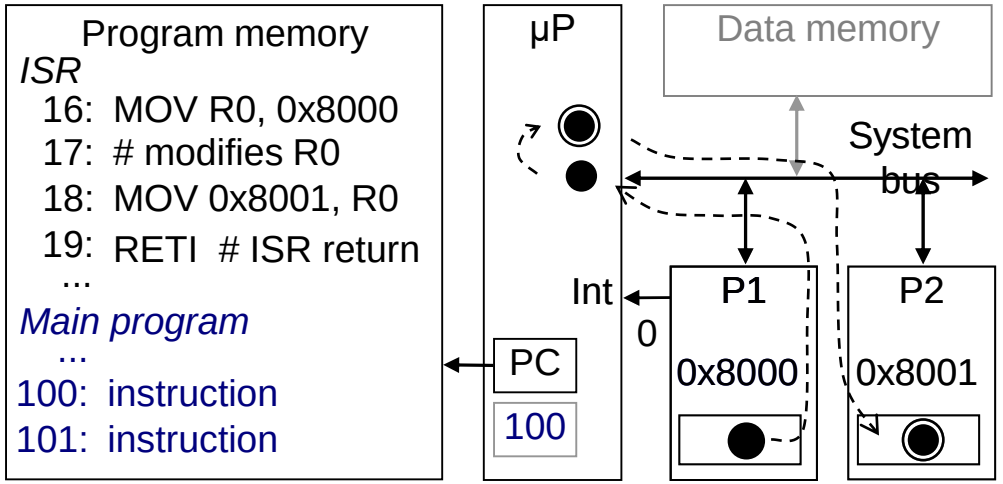
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-Driven I/O using Fixed ISR Location

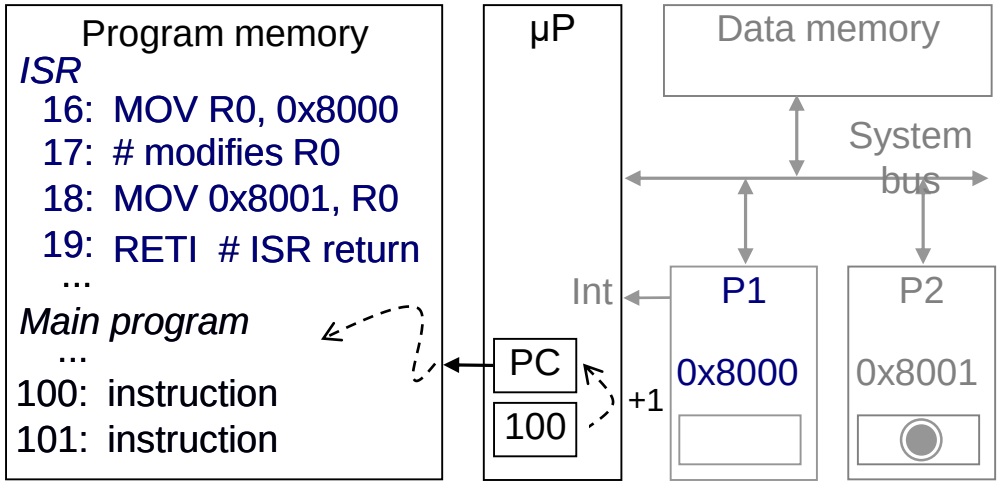
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

4(b): After being read, P1 deasserts *Int*.

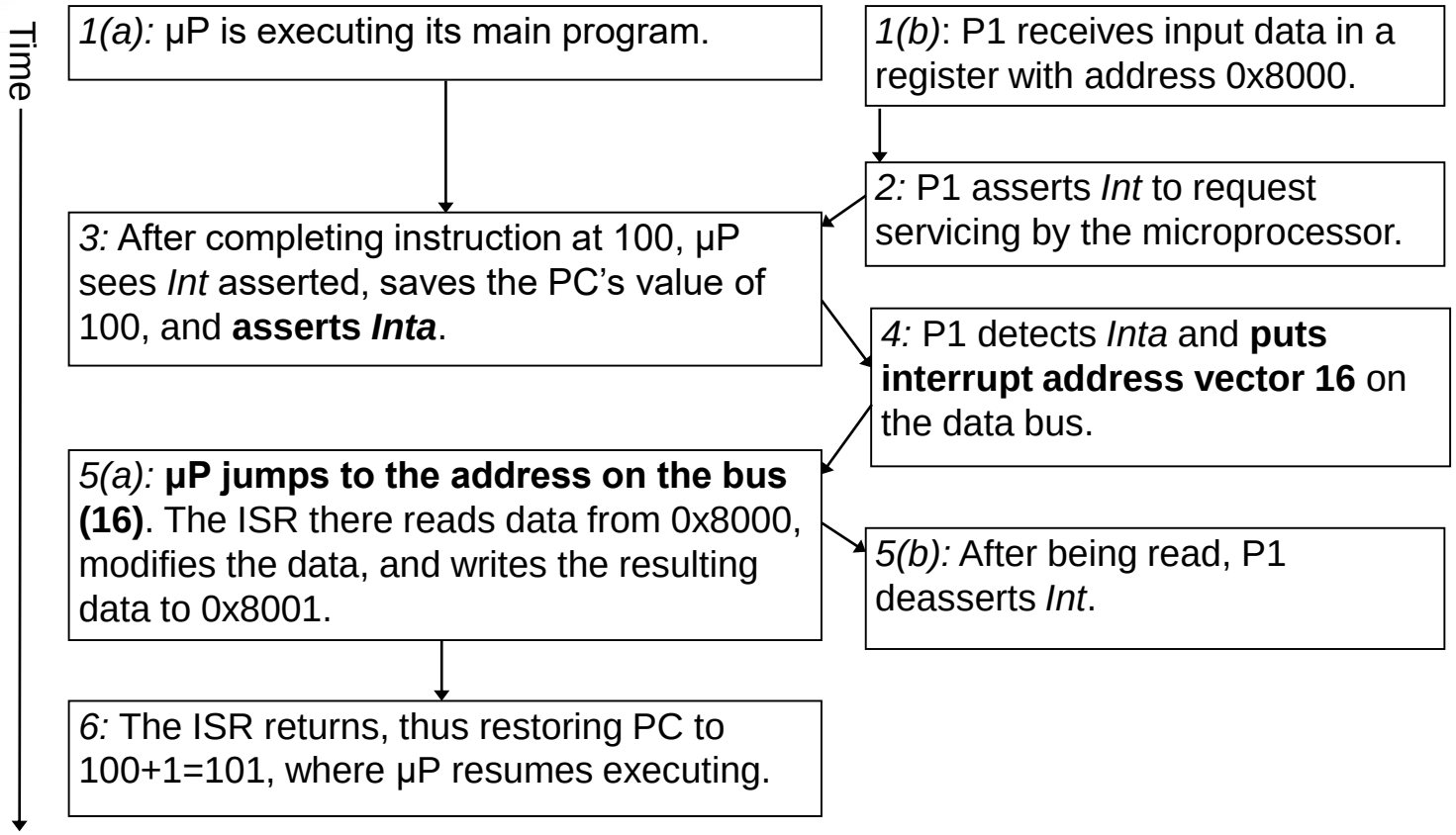


Interrupt-Driven I/O using Fixed ISR Location

5: The ISR returns, thus restoring PC to 100+1=101, where μP resumes executing.



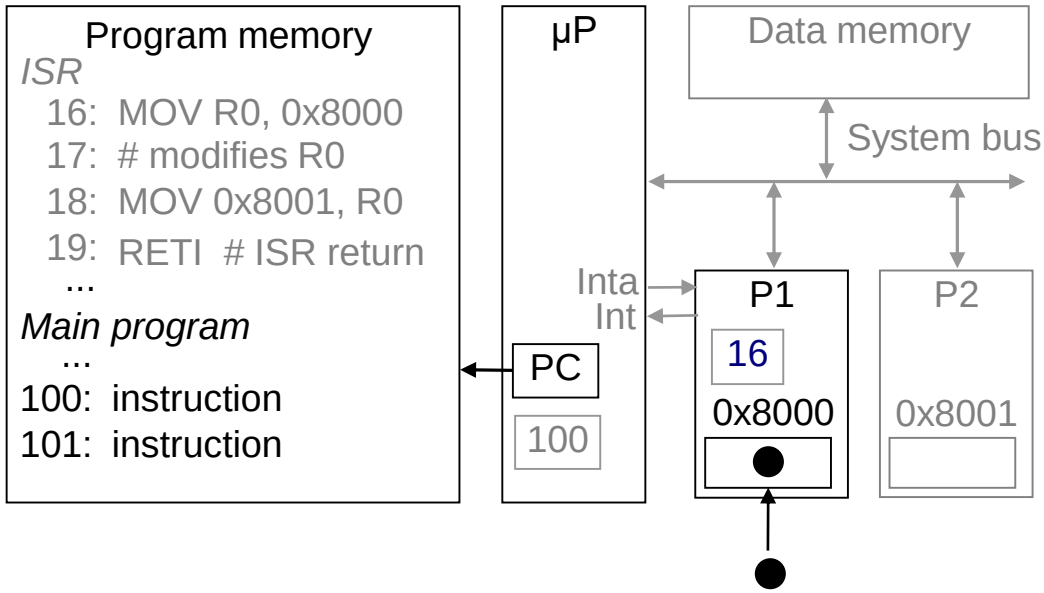
Interrupt-Driven I/O using Vectored Interrupt



Interrupt-Driven I/O using Vectored Interrupt

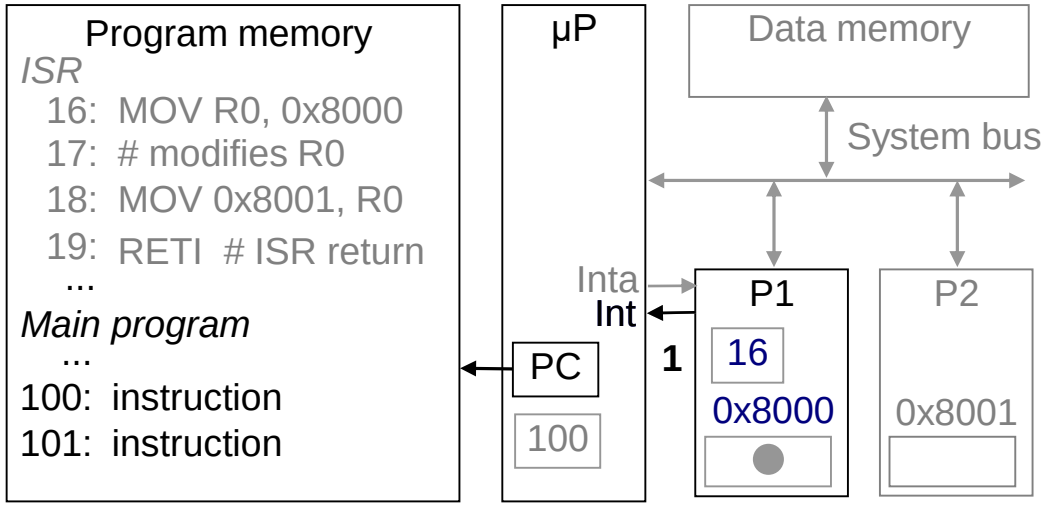
1(a): μP is executing its main program

1(b): P1 receives input data in a register with address 0x8000.



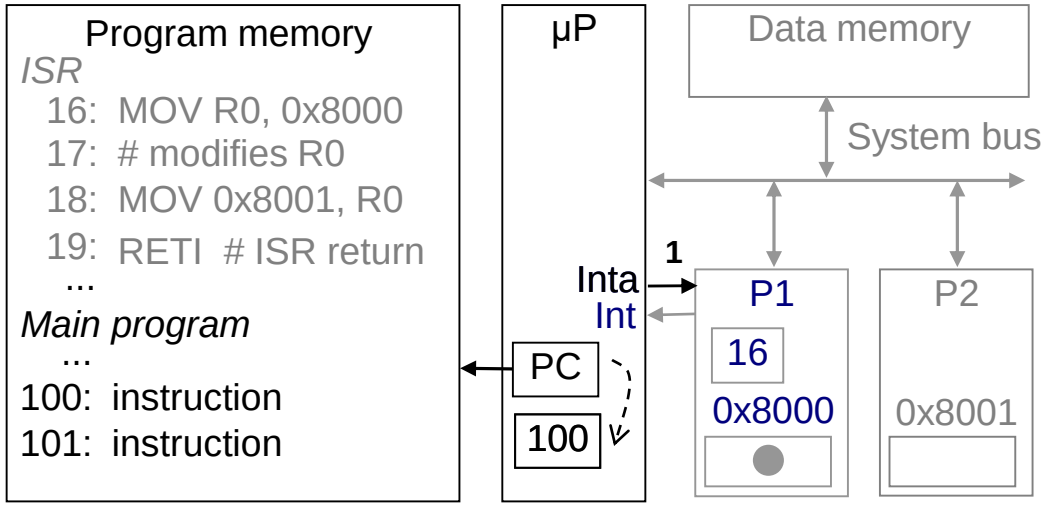
Interrupt-Driven I/O using Vectored Interrupt

2: P1 asserts *Int* to request servicing by the microprocessor



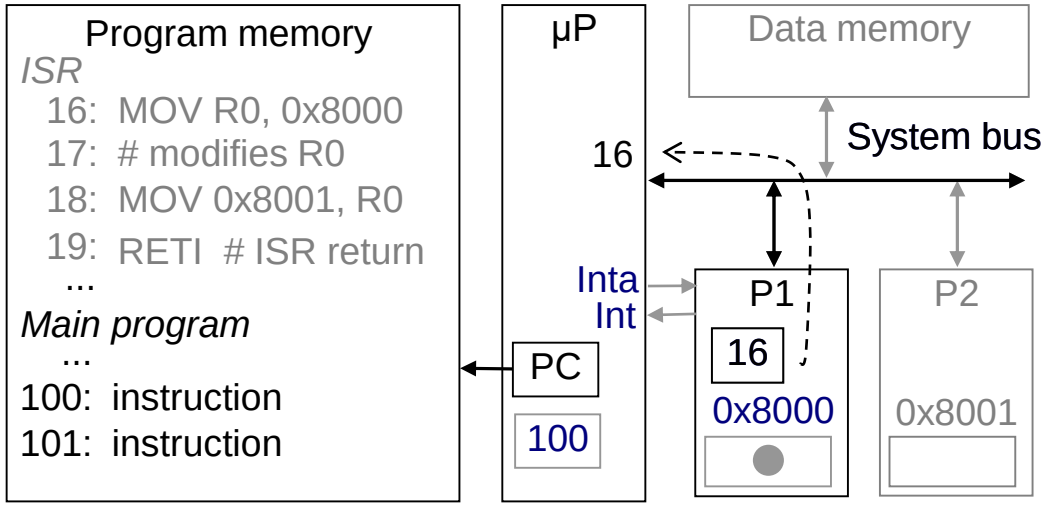
Interrupt-Driven I/O using Vectored Interrupt

3: After completing instruction at 100, µP sees *Int* asserted, saves the PC's value of 100, and **asserts *Inta***



Interrupt-Driven I/O using Vectored Interrupt

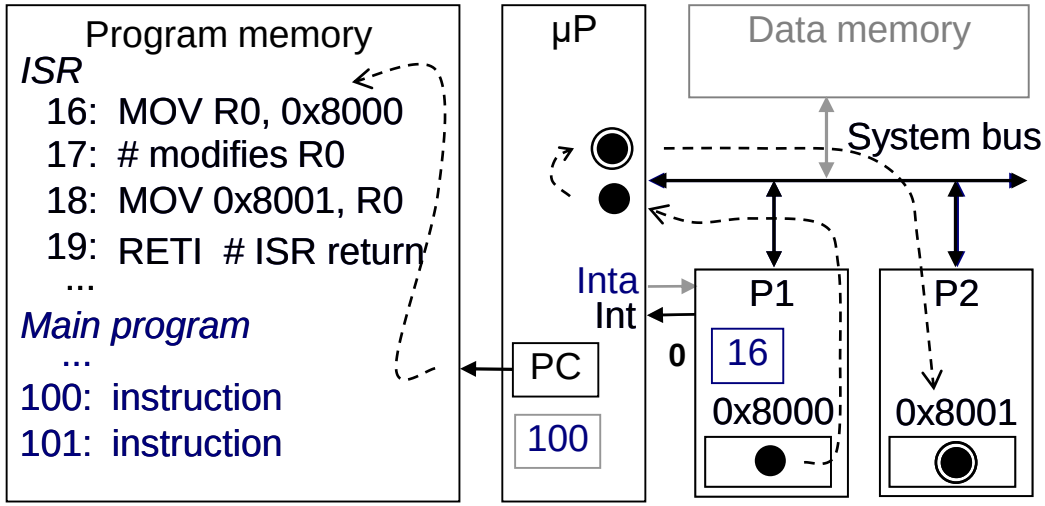
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-Driven I/O using Vectored Interrupt

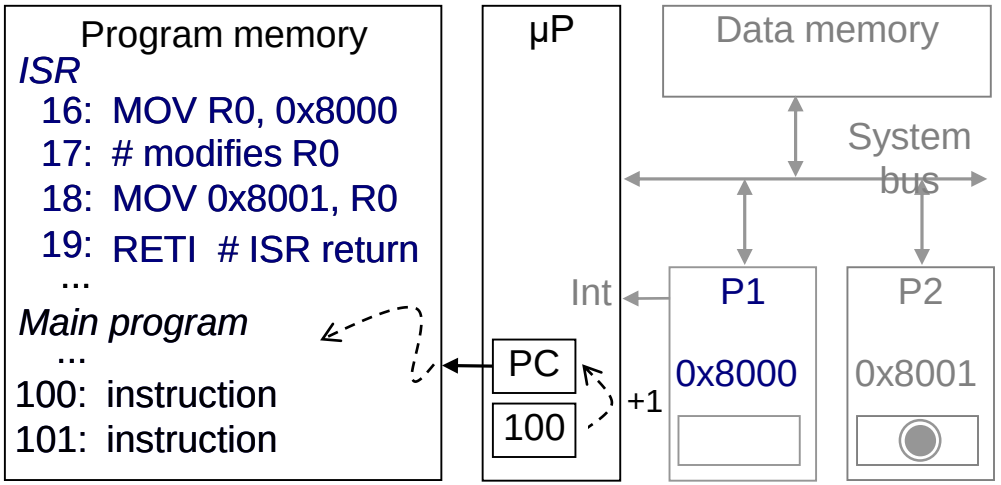
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

5(b): After being read, P1 deasserts *Int*.



Interrupt-Driven I/O using Vectored Interrupt

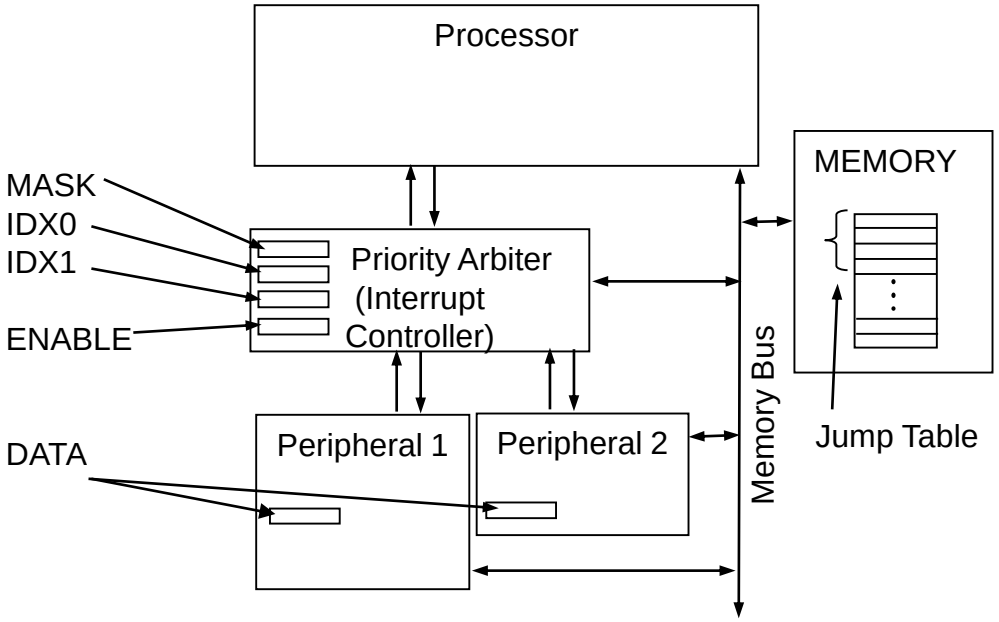
6: The ISR returns, thus restoring the PC to 100+1=101, where the μP resumes



Interrupt Address Table

- Compromise between fixed and vectored interrupts
  - One interrupt pin
  - Table in memory holding ISR addresses (maybe 256 words)
  - Peripheral doesn't provide ISR address, but rather index into table
    - Fewer bits are sent by the peripheral
    - Can move ISR location without changing peripheral

Interrupt Table



- Fixed priority: i.e., Peripheral1 has the highest priority
- Keyword "_at_" followed by memory address forces compiler to place variables in specific memory locations
  - e.g., memory-mapped registers in arbiter (interrupt controller), peripherals
- A peripheral's index into interrupt table is sent to memory-mapped register in arbiter **(interrupt controller)**
- Peripherals receive external data and raise interrupt

Interrupt Table

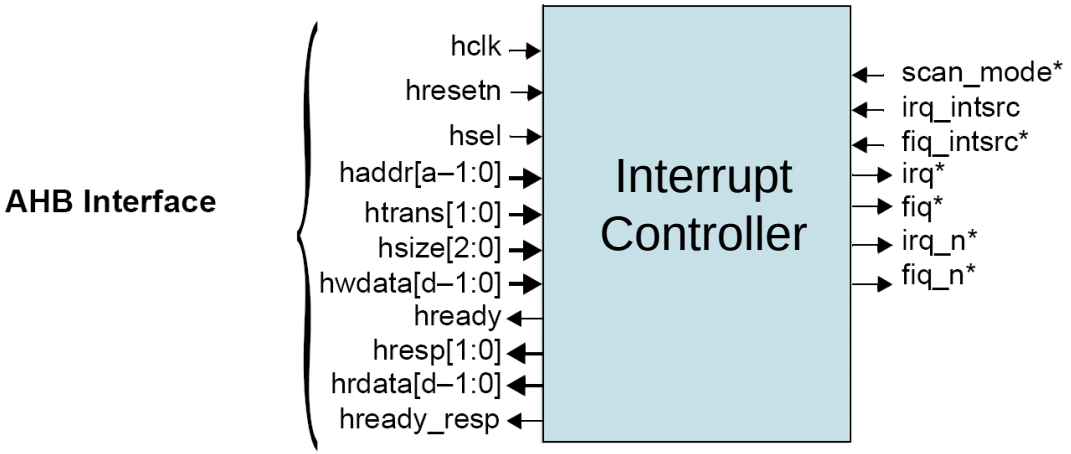
```
unsigned char ARBITER_MASK_REG              _at_ 0xfff0;
unsigned char ARBITER_CH0_INDEX_REG         _at_ 0xfff1;
unsigned char ARBITER_CH1_INDEX_REG         _at_ 0xfff2;
unsigned char ARBITER_ENABLE_REG            _at_ 0xfff3;
unsigned char PERIPHERAL1_DATA_REG          _at_ 0xffe0;
unsigned char PERIPHERAL2_DATA_REG          _at_ 0xffe1;
unsigned void* INTERRUPT_LOOKUP_TABLE[256]  _at_ 0x0100;




    void main() {
                  InitializePeripherals();
                     for(;;) {} // main program goes here
}
```

Interrupt Table

```
void Peripheral1_ISR(void) {
        unsigned char data;
         data = PERIPHERAL1_DATA_REG;
         // do something with the data
}
   void Peripheral2_ISR(void) {
        unsigned char data;
         data = PERIPHERAL2_DATA_REG;
         // do something with the data
}
    void InitializePeripherals(void) {
          ARBITER_MASK_REG = 0x03;      // enable both channels
          ARBITER_CH0_INDEX_REG = 13;
          ARBITER_CH1_INDEX_REG = 17;
             INTERRUPT_LOOKUP_TABLE[13] = (void*)Peripheral1_ISR;
             INTERRUPT_LOOKUP_TABLE[17] = (void*)Peripheral2_ISR;
        ARBITER_ENABLE_REG = 1;
}
```

Interrupt Controller

- A Slave device
- Support
  - multiple interrupt sources
  - Vectored interrupt
  - Software interrupt
  - Priority filtering
  - Masking
  - Programmable for some cases



x = number of slave selects that the slave requires
a = Width of address bus
d = Width of data bus, which is same width as AHB_DATA_WIDTH
* = optional signals

Additional Interrupt Issues

- Maskable vs. non-maskable interrupts
  - **Maskable:** programmer can set bit that causes processor to ignore interrupt
    - Important when in the middle of time-critical code
  - **Non-maskable:** a separate interrupt pin that can't be masked
    - Typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory

Additional Interrupt Issues

- Jump to ISR
  - Some microprocessors treat jump the same as call of any subroutine
    - Complete state saved (PC, registers) – may take hundreds of cycles
  - Others only save partial state, like PC only
    - Thus, ISR must not modify registers, or else must save them first
    - Assembly-language programmer must be aware of which registers stored

Sources of Interrupt Overhead

- Handler execution time
- Interrupt mechanism overhead
- Register save/restore
- Pipeline-related penalties
- Cache-related penalties

ARM Interrupts

- ARM7 supports two types of interrupts:
  - Fast interrupt requests (FIQs).
  - Interrupt requests (IRQs).
- Interrupt vector address
  - FIQ: 0x0000001C
  - IRQ: 0x00000018

ARM Interrupt Procedure

- CPU actions:
  - Save PC. Copy CPSR (current program status register) to SPSR (saved program status register)
  - Force bits in CPSR to record interrupt
  - Force PC to vector
- Handler responsibilities:
  - Restore proper PC
  - Restore CPSR from SPSR
  - Clear interrupt disable flags

ARM Interrupt Latency

- Worst-case latency to respond to FIQ is **28 cycle**:
  - Three cycles to synchronize external request
  - Up to 20 cycles to complete current instruction
  - Three cycles for data abort
  - Two cycles to enter interrupt handling state
- The best case is 4 cycle

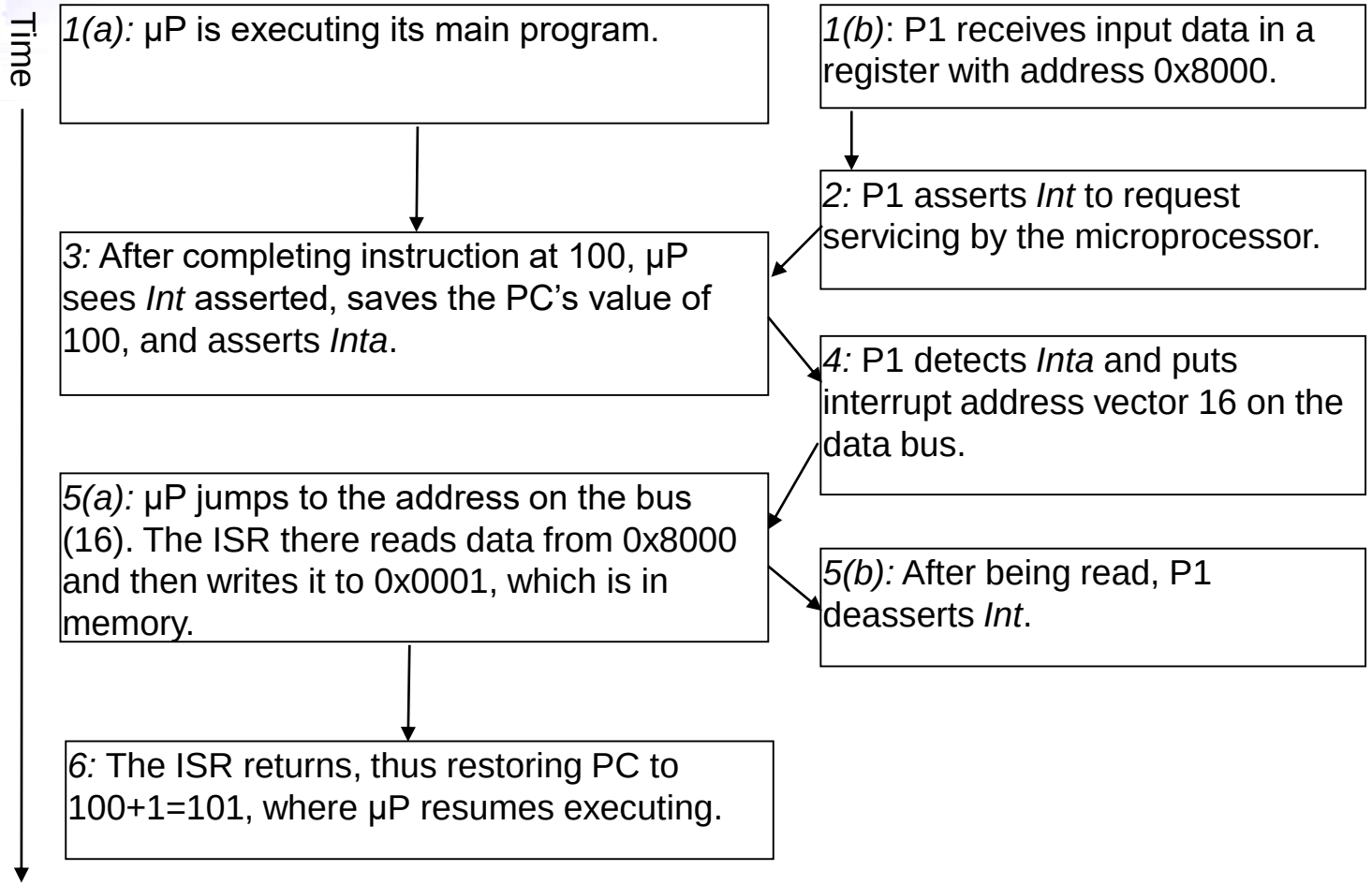
Direct Memory Access (DMA)

- Buffering
  - Temporarily storing data in memory before processing
  - Data accumulated in peripherals commonly buffered
- Microprocessor could handle this with ISR
  - Storing and restoring microprocessor state (interrupt overhead) is inefficient
  - Regular program must wait
- DMA controller is more efficient
  - Separate single-purpose processor
  - Microprocessor relinquishes control of system bus to DMA controller

Direct Memory Access (DMA)

- Microprocessor can meanwhile execute its regular program
  - No inefficient storing and restoring state due to ISR call
  - Regular program needs not to wait unless it requires the system bus
    - **Harvard architecture** – processor can fetch and execute instructions as long as they don't access data memory – if they do, processor stalls
    - A system with separate bus between the microprocessor and **cache (or TCM)** may be able to execute when DMA is working

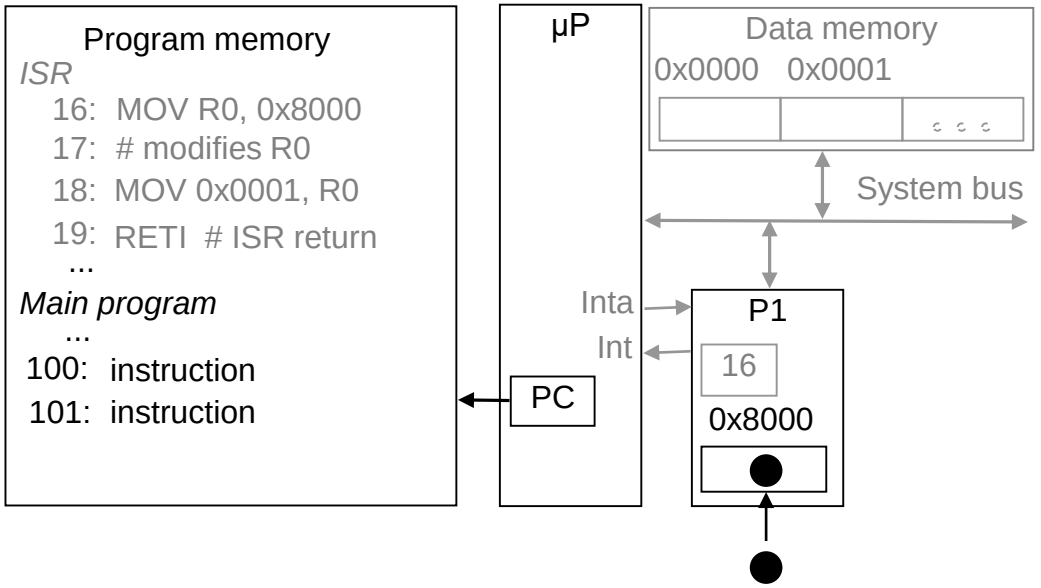
Peripheral to Memory Transfer *without* DMA



Peripheral to Memory Transfer *without* DMA

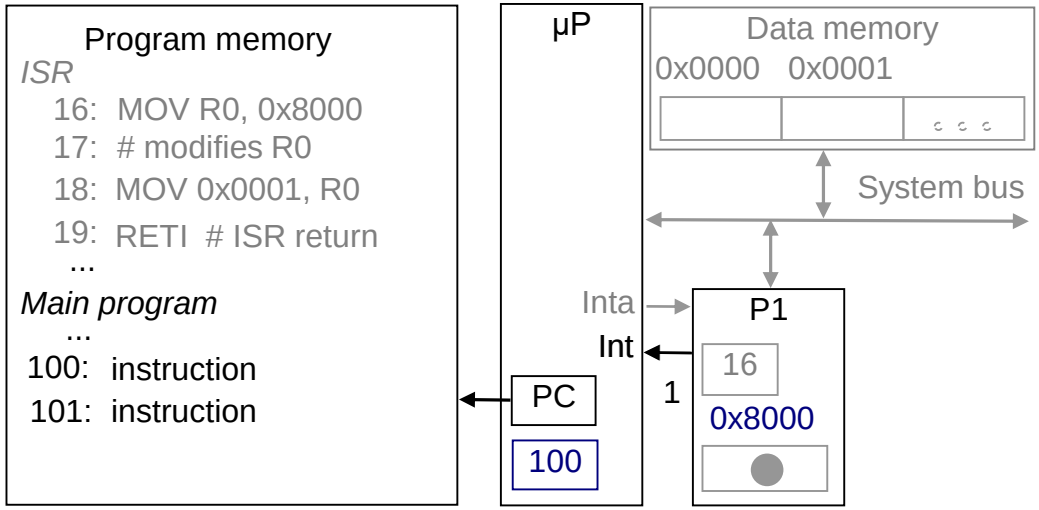
1(a): μP is executing its main program

1(b): P1 receives input data in a register with address 0x8000.



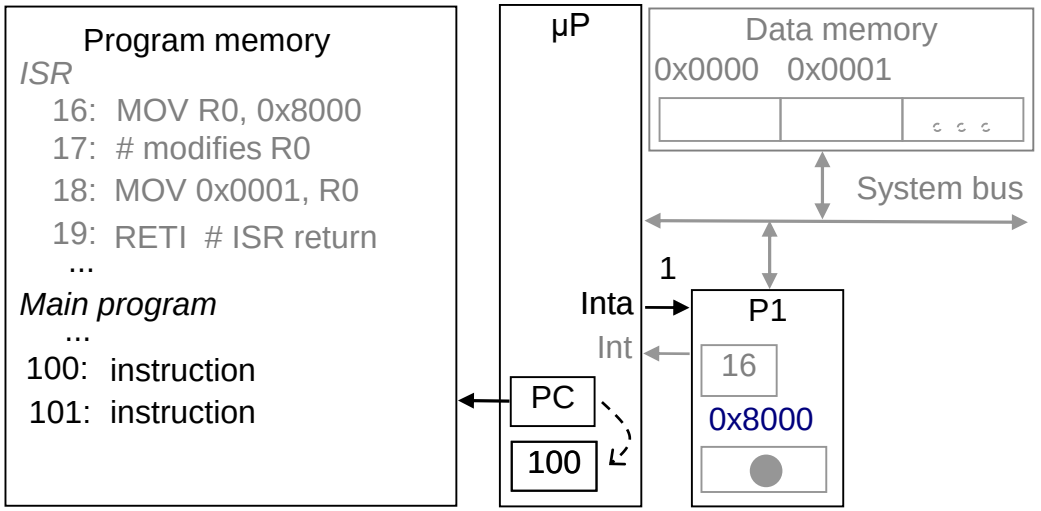
Peripheral to Memory Transfer *without* DMA

2: P1 asserts *Int* to request servicing by the microprocessor



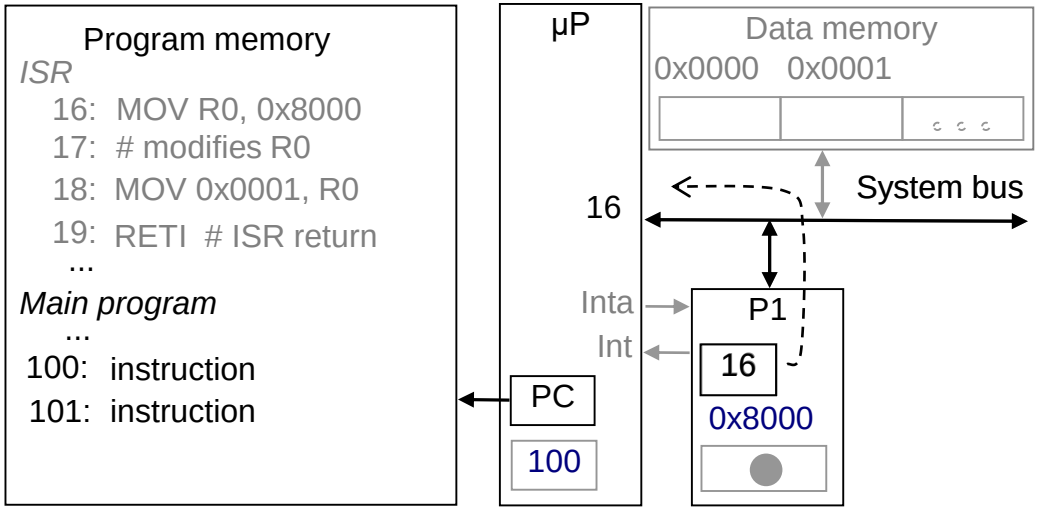
Peripheral to Memory Transfer *without* DMA

3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and asserts *Inta*.



Peripheral to Memory Transfer *without* DMA

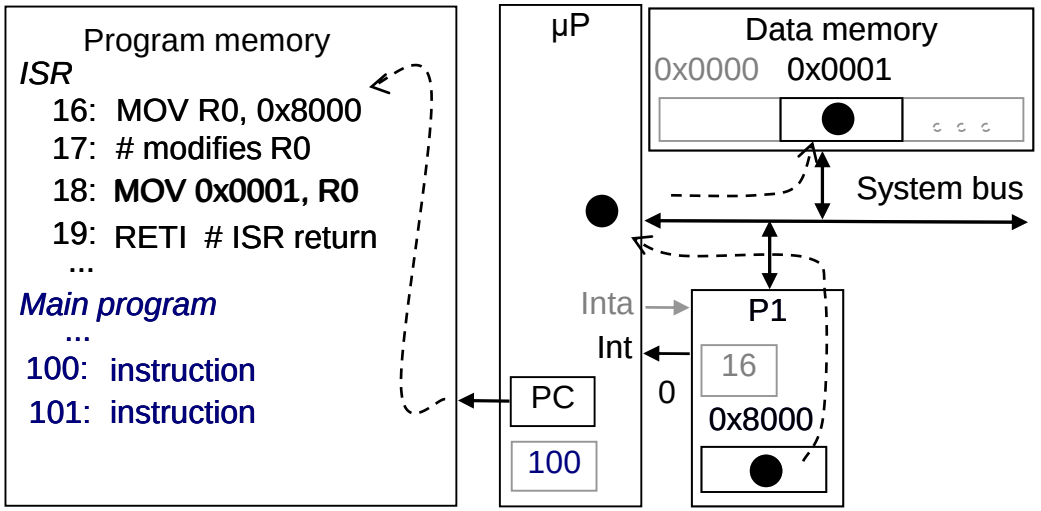
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



Peripheral to Memory Transfer *without* DMA

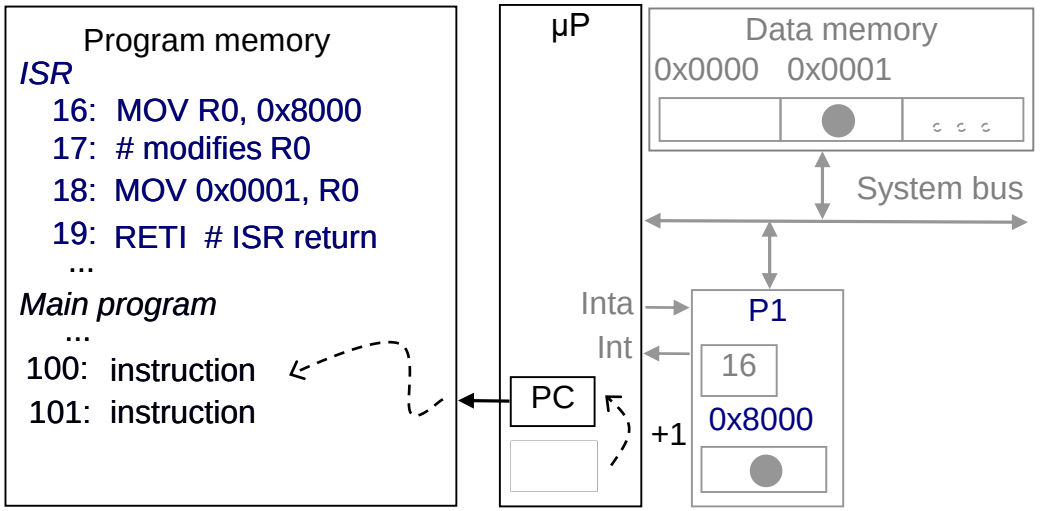
5(a): μP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.

5(b): After being read, P1 de-asserts *Int*.

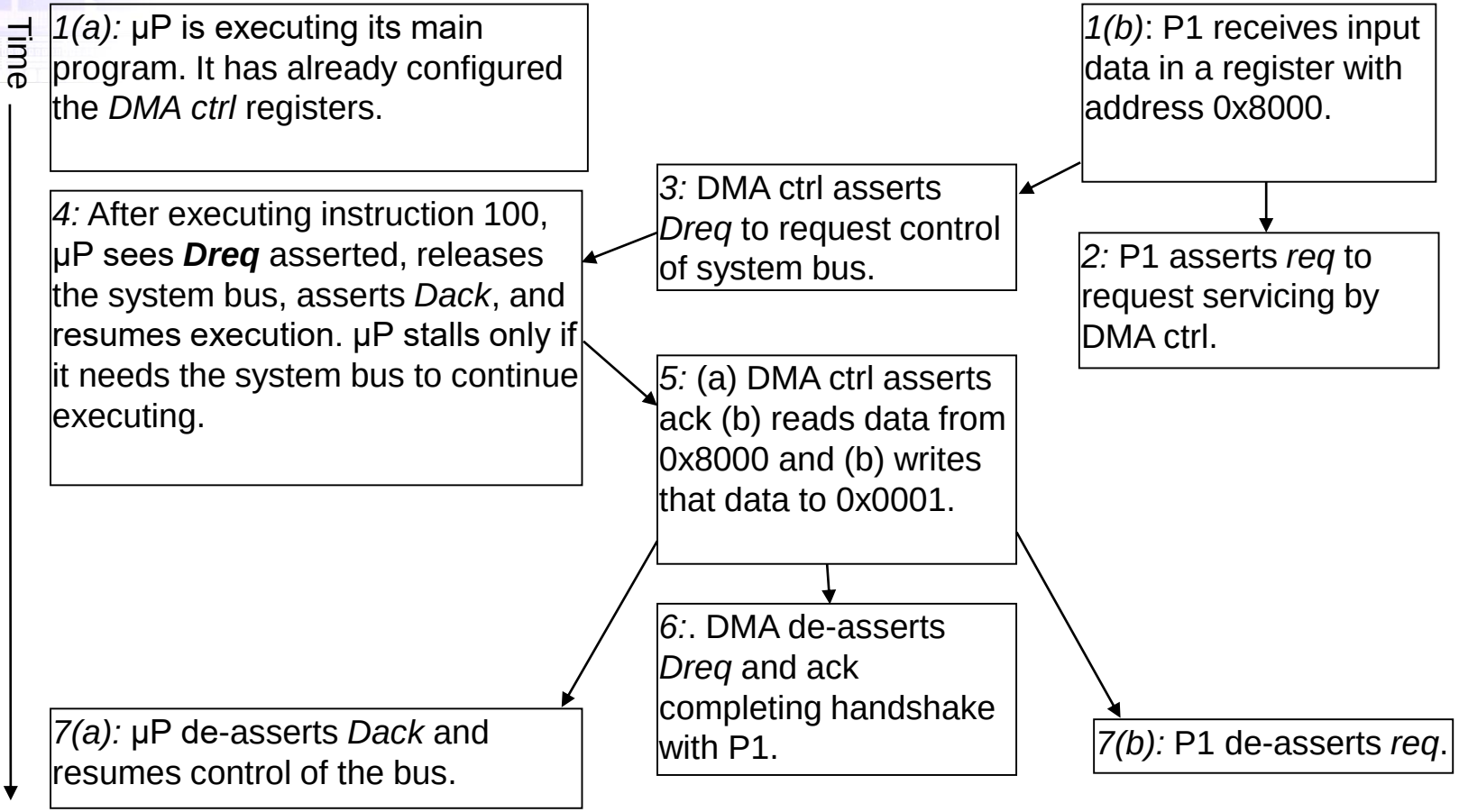


Peripheral to Memory Transfer *without* DMA

6: The ISR returns, thus restoring PC to 100+1=101, where μP resumes executing.



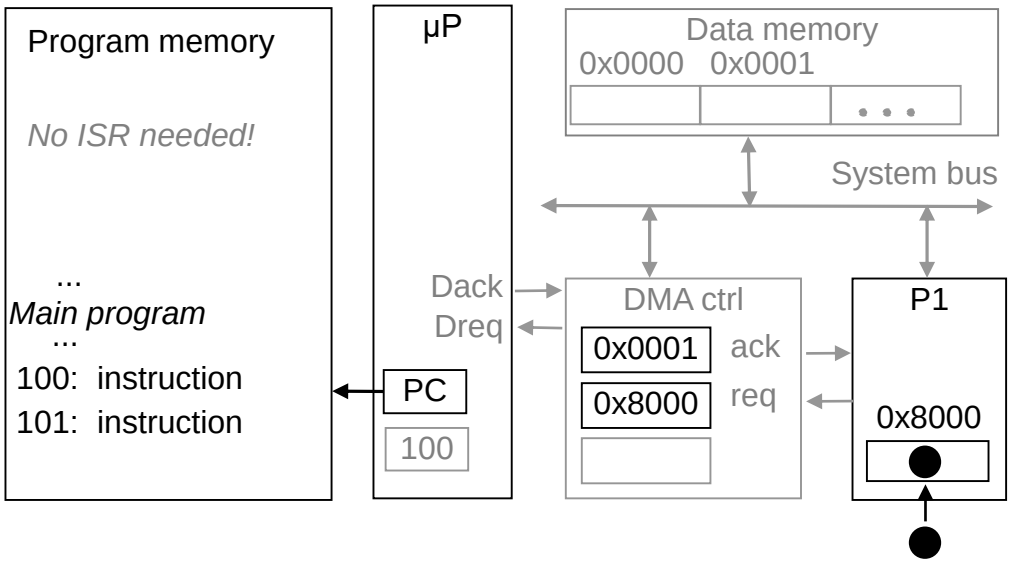
Peripheral to Memory Transfer with DMA



Peripheral to Memory Transfer with DMA

1(a): μP is executing its main program. It has already configured the DMA ctrl registers

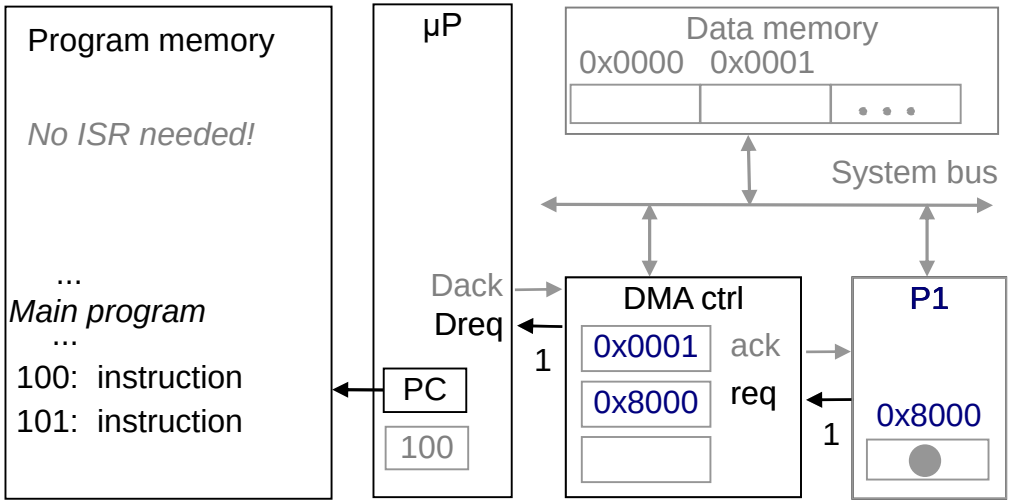
1(b): P1 receives input data in a register with address 0x8000.



Peripheral to Memory Transfer with DMA

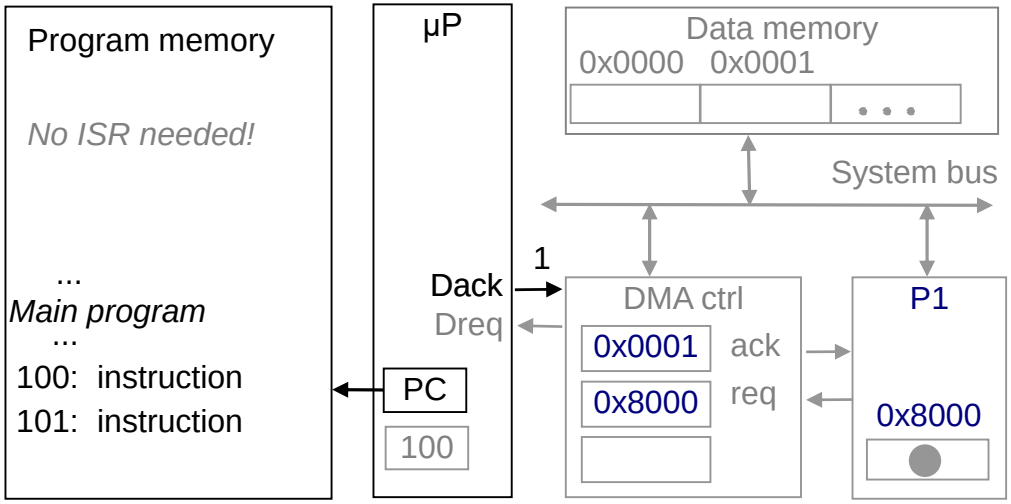
2: P1 asserts *req* to request servicing
by DMA ctrl.

3: DMA ctrl asserts *Dreq* to request control of system bus



Peripheral to Memory Transfer with DMA

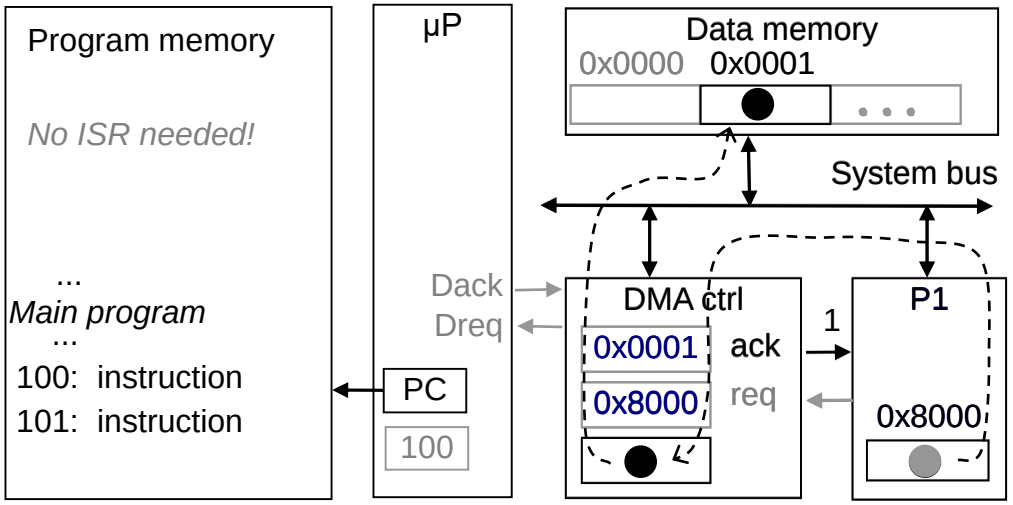
4: After executing instruction 100, μP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, μP stalls only if it needs the system bus to continue executing.



Peripheral to Memory Transfer with DMA

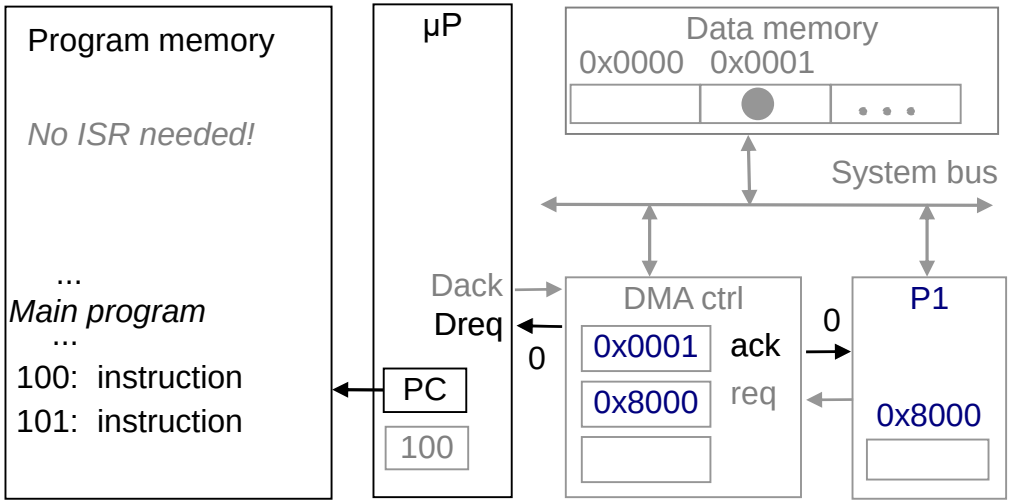
5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.

(Meanwhile, processor still executing if not stalled!)



Peripheral to Memory Transfer with DMA

6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.



Arbitration: Priority Arbiter

- Consider the situation where multiple peripherals request service from single resource (e.g., microprocessor, DMA controller, memory controller) simultaneously - which gets serviced first?
- Arbiter
  - Single-purpose processor
  - Peripherals make requests to arbiter, arbiter makes requests to resource
  - Arbiter connected to system bus for configuration only
- Priority arbiter
  - The arbiter grants the request according to a priority list

Arbitration

- The arbitration process plays a crucial role in determining the performance of the system
- It assigns the priorities with which processor are granted the access to the shared communication resource
- Arbitration become more and more important
  - Increasing integration levels of SoC → increase contention → violate real-time constraints → need efficient contention resolution scheme

Arbitration: Priority Arbiter

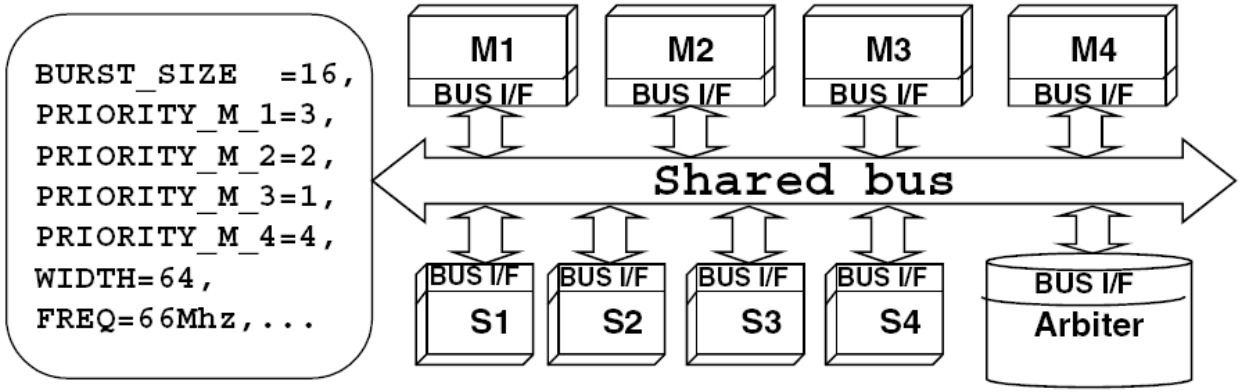
- Types of priority
  - Fixed priority
  - Time division multiple access (TDMA)
  - Rotating priority (round-robin)
    - Priority changed based on history of servicing
    - Better distribution of servicing especially among peripherals with similar priority demands
  - Slot reservation
  - LOTTERYBUS

Better Arbiter?

- A good arbiter should be able to provide
  - Proportional allocation of communication bandwidth
  - Low latency communication for high priority data transfer

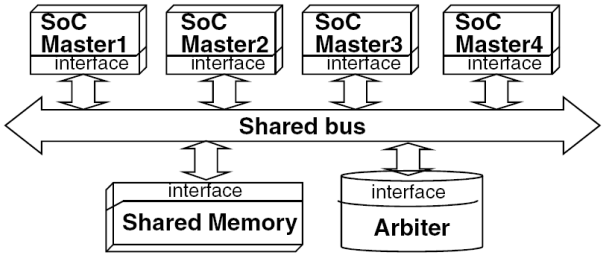
Fixed Priority

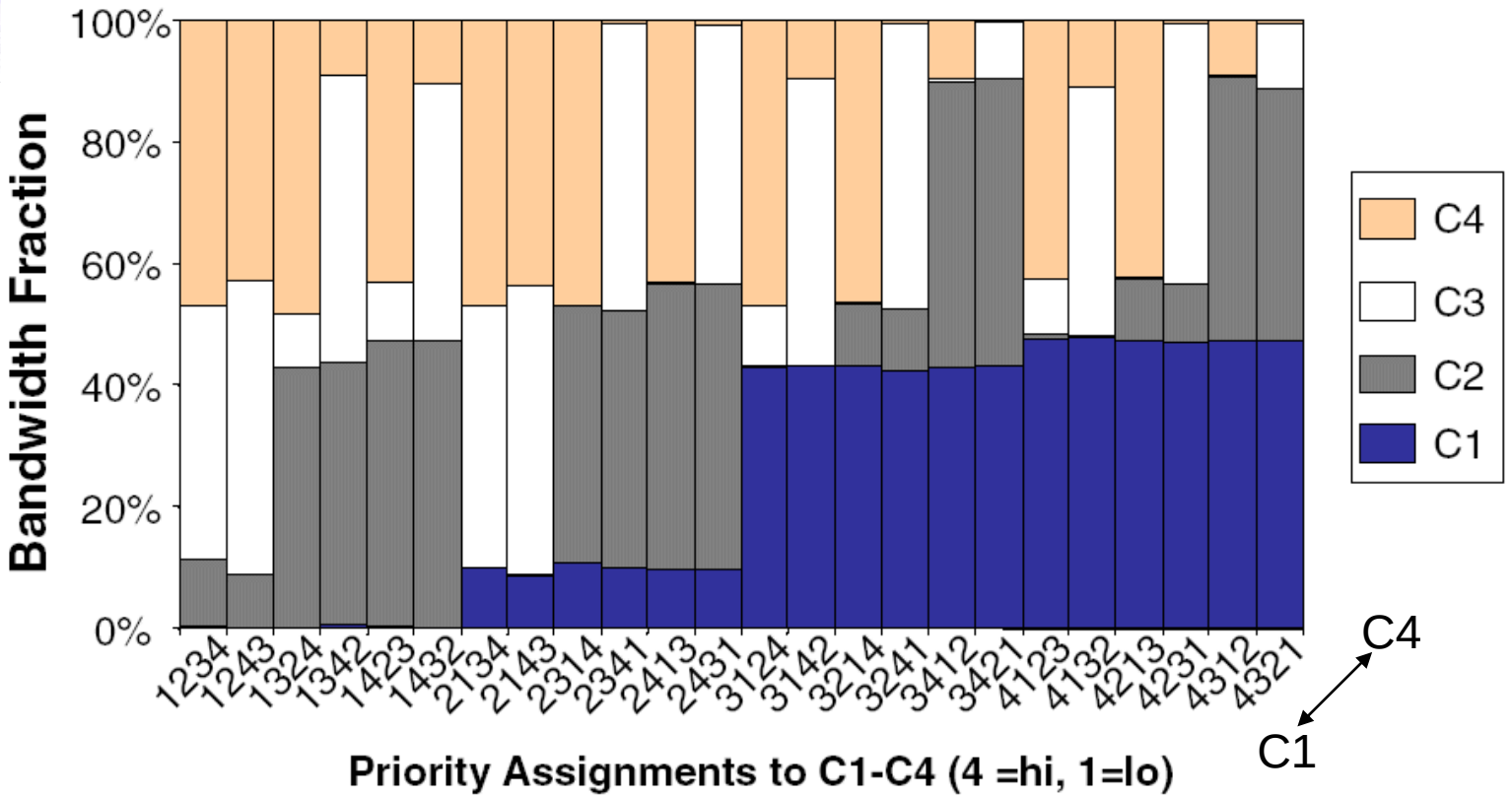
- Each peripheral has a unique rank
- Highest rank is chosen first with simultaneous requests
- Preferred when clear difference in rank between peripherals
- May lead to starvation for the low priority components



Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs,” DAC 2001.

Fixed Priority





Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs,” DAC 2001.

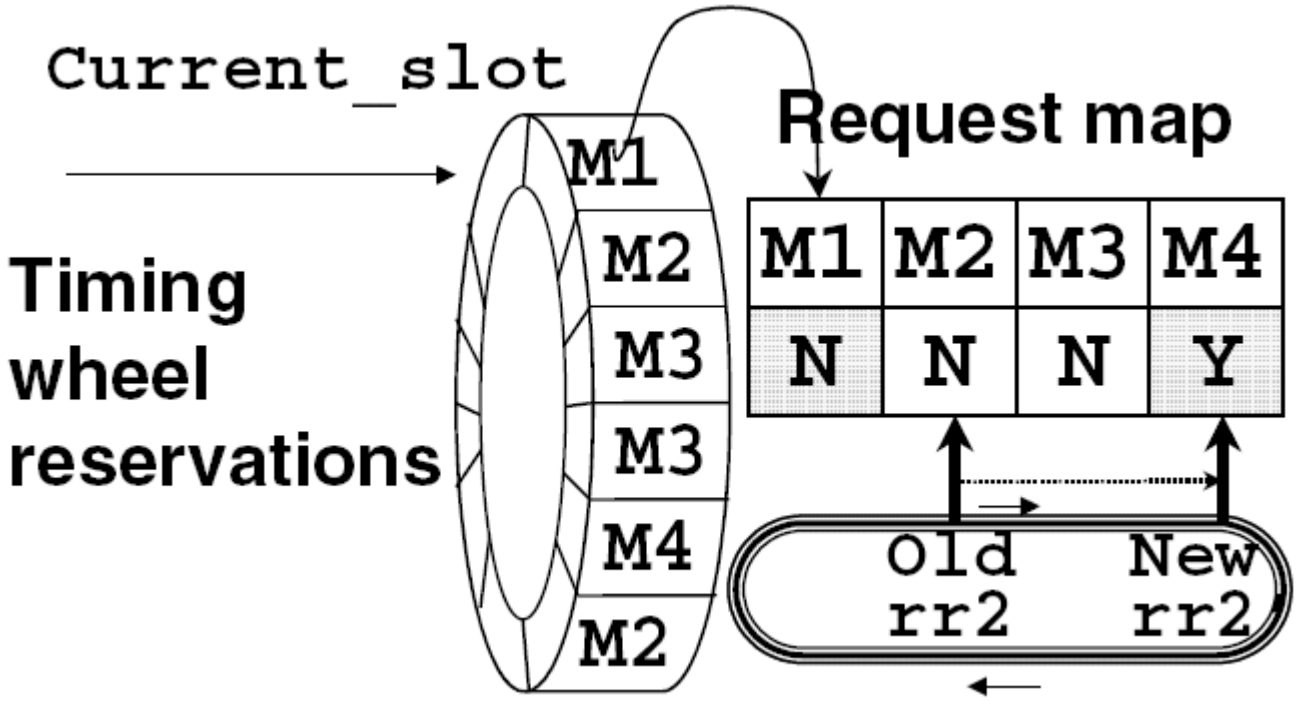
Fixed Priority

- The fraction of bandwidth a component receives is extremely sensitive to the priority value it is assigned
- Low priority component get a negligible fraction of the bus bandwidth → **leads to starvation for the low priority component**

Time Division Multiple Access (TDMA)

- Guarantee bandwidth for each component
- Long latencies for high priority components
- Sometimes, two-level arbitration protocol is used:
  - Timing wheel, each slot is statically reserved for a unique master
  - To alleviate the problem of wasted slots, a second level of arbitration issues a grant to the next requesting master in a round-robin fashion if the assigned master does not have a pending communication request

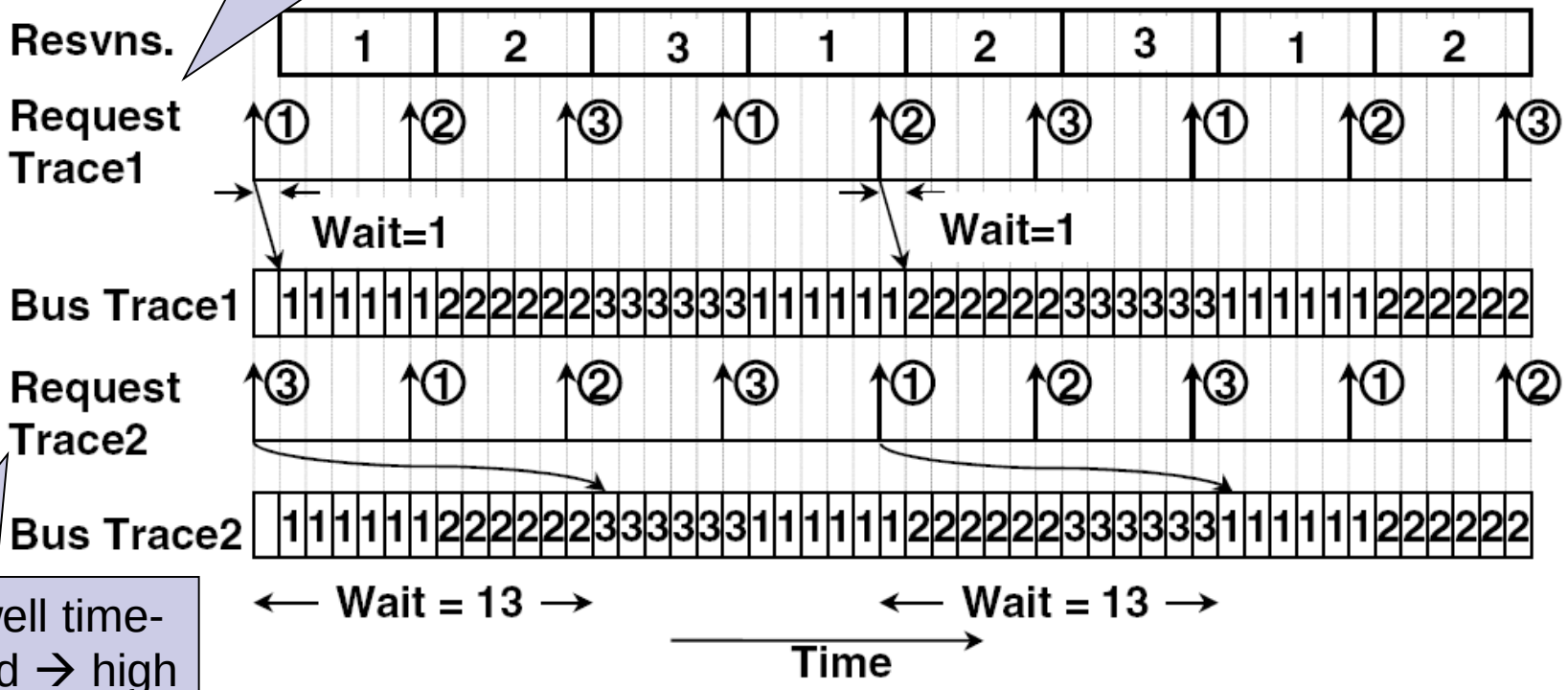
Time Division Multiple Access (TDMA)



Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs,” DAC 2001.

TDMA

Well time-aligned → low latency



Not well time-aligned → high latency

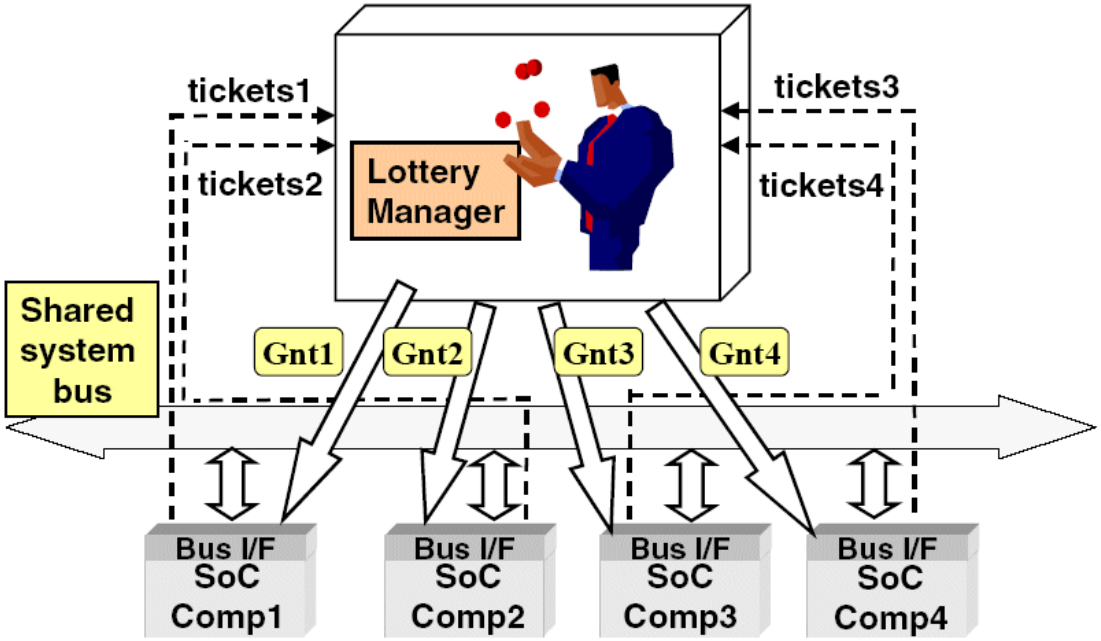
Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, "LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs," DAC 2001.

Rotating Priority (Round-Robin)

- Priority changed based on history of servicing
- Better distribution of servicing especially among peripherals with similar priority demands
- High bus utilization
- Worst-case waiting time is reliably predictable
- The actual bandwidth is uncertain

LOTTERYBUS Communication Architecture

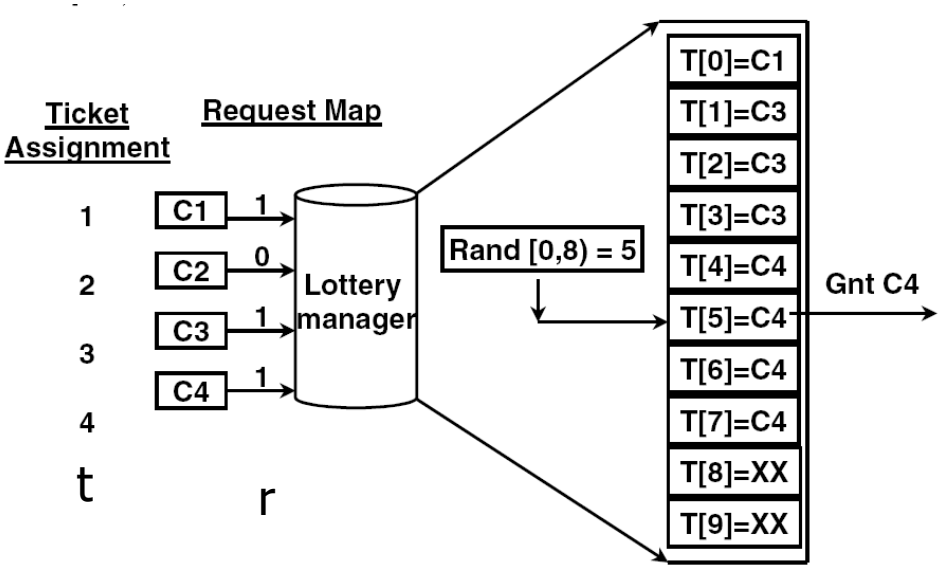
- Lottery manager
  - Randomly choose one master to be the winner of the lottery
- A maximum transfer size limits to prevent a master from monopolizing the bus



Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs,” DAC 2001.

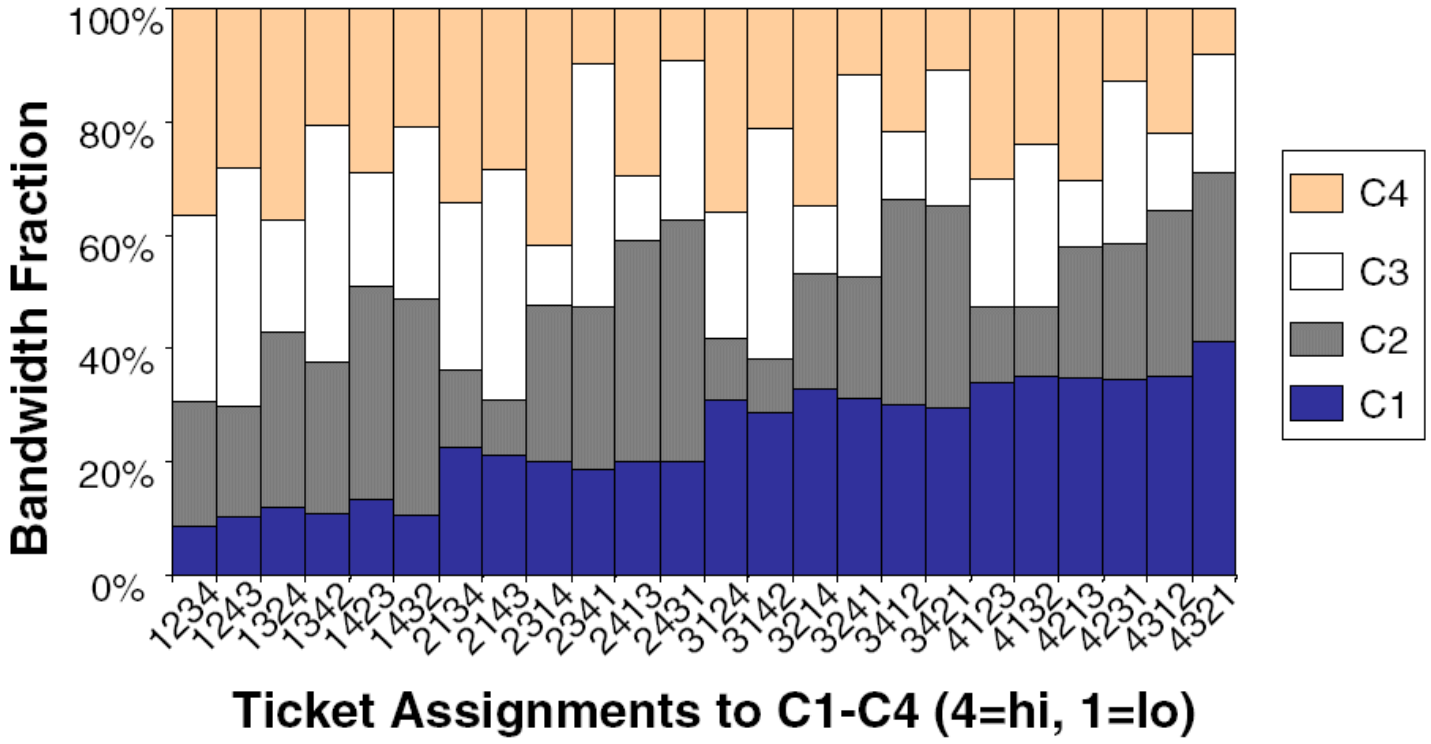
LOTTERYBUS Communication Architecture

- The master to be granted is chosen in a randomized way with probability of granting component.



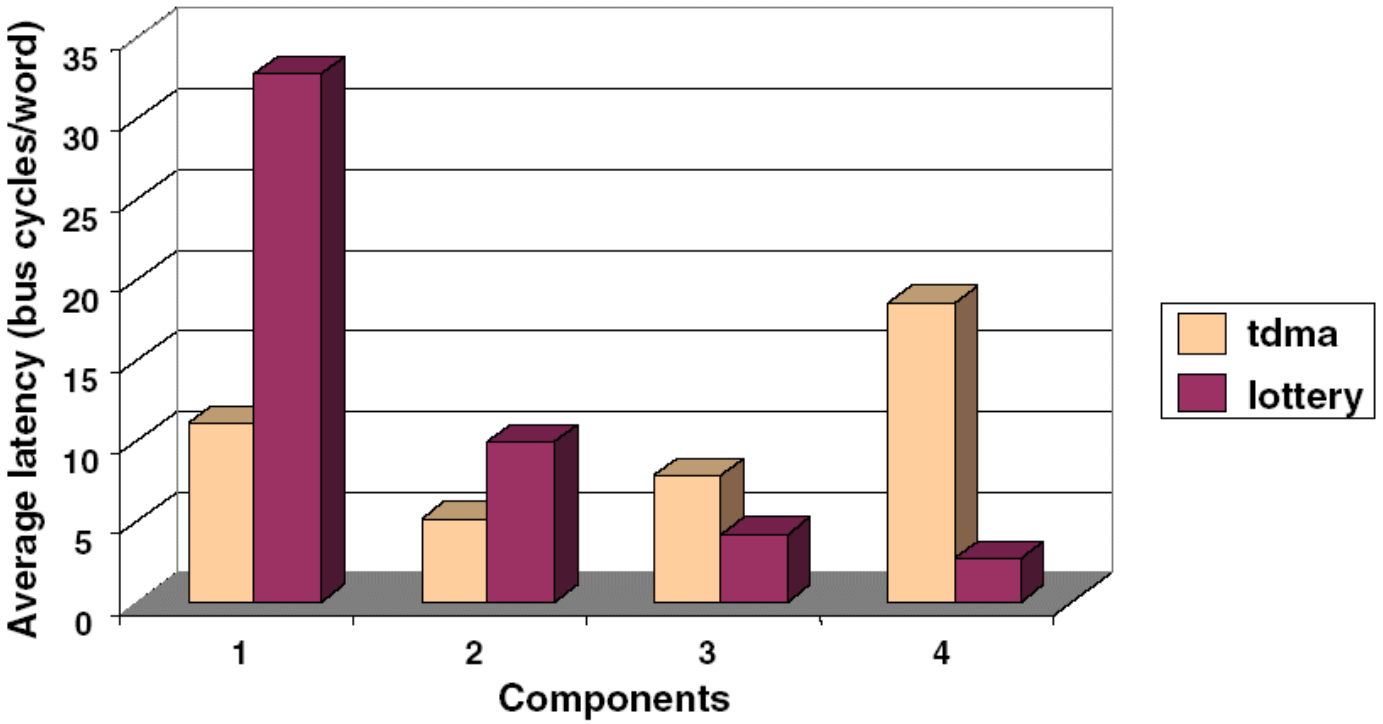
Probability of grating component C_i: $$P(C_i) = \frac{r_i \cdot t_i}{\sum_{j=1}^{n} r_j \cdot t_j}$$

LOTTERYBUS: Bandwidth Allocation



Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs,” DAC 2001.

LOTTERYBUS: Latency



Source: K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “LOTTERYBUS, a new high-performance communication architecture for System-on-Chip designs,” DAC 2001.

Slot Reservation

- TDMA + round-robin
- Only one master is periodically allocated a slot for the contention-free access
  - The length of the time slot is adjustable
- For the inter-slot time, the contention among the remaining masters is managed in a round-robin fashion

Source: F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo, “Performance analysis of arbitration policies for SoC communication architectures,” *Design Automation for Embedded Systems*, vol. 8, pp. 189—210, 2003.

Important Notes

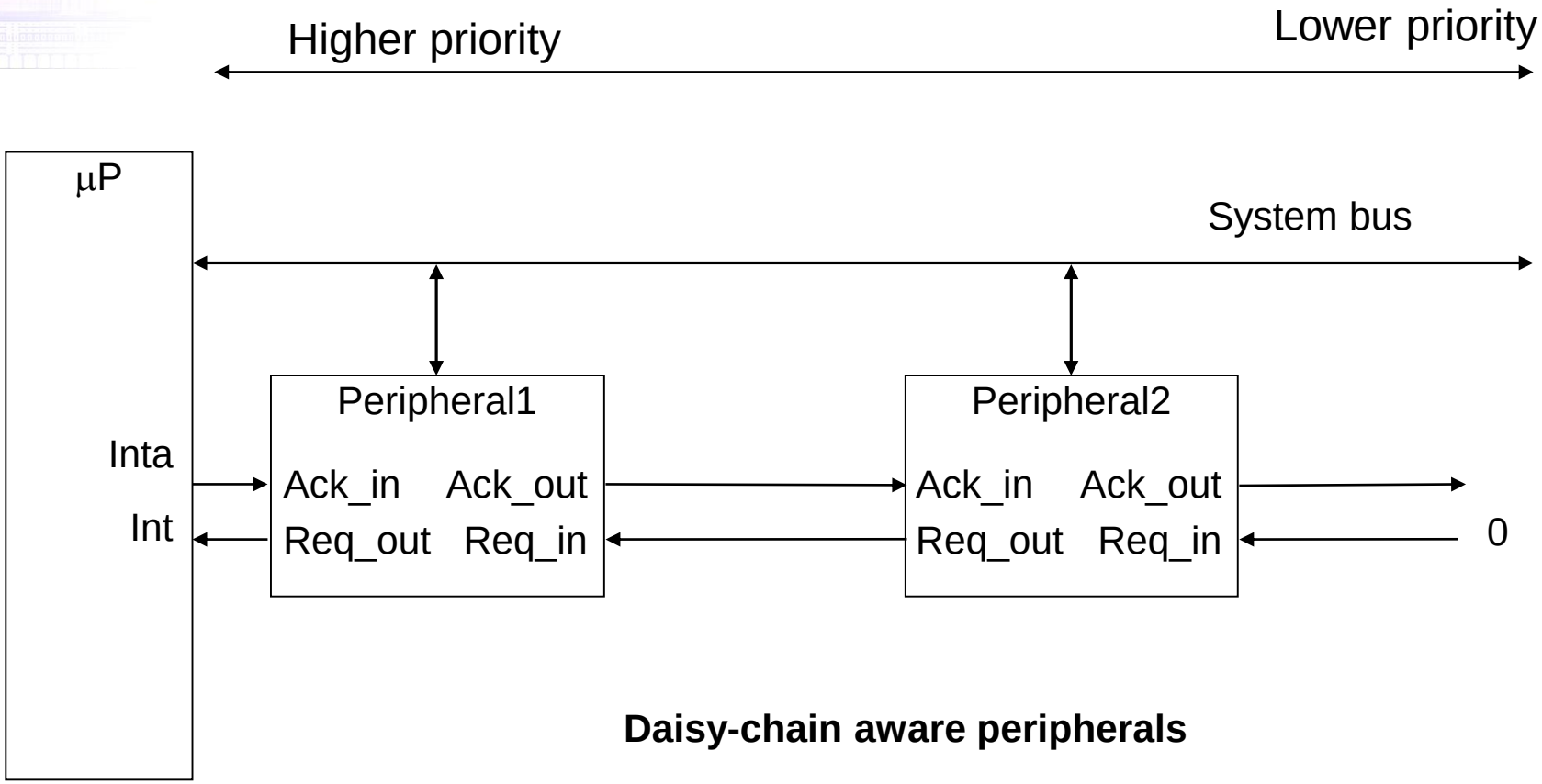
- The optimal bus arbitration policy is not unique, but strongly depends on the traffic conditions
- There exists a trade-off between contention-avoidance bus arbitration policies and contention-resolution bus protocols

Source: F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo, “Performance analysis of arbitration policies for SoC communication architectures,” *Design Automation for Embedded Systems*, vol. 8, pp. 189—210, 2003.

Arbitration: Daisy-Chain Arbitration

- Arbitration is done by peripherals
  - Built into peripheral or external logic added
    - *req* input and *ack* output are added to each peripheral
- Peripherals are connected to each other in daisy-chain manner
  - One peripheral connected to resource, all others connected “upstream”
  - Peripheral’s *req* flows “downstream” to resource, resource’s *ack* flows “upstream” to requesting peripheral
  - Closest peripheral has highest priority

Arbitration: Daisy-Chain Arbitration



Daisy-chain aware peripherals

Arbitration: Daisy-Chain Arbitration

- Pros/cons
  - Easy to add/remove peripheral - no system redesign needed
  - Does not support rotating priority
  - One broken peripheral can cause the loss of accessing to other peripherals

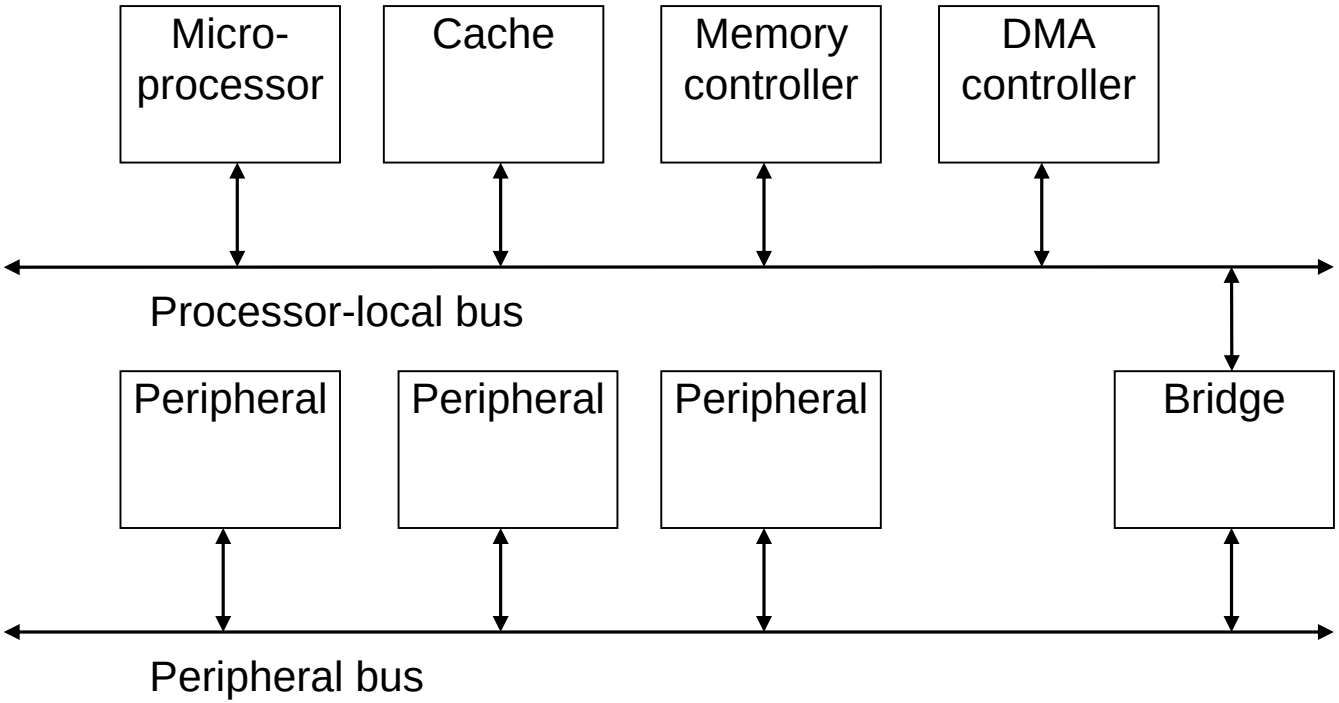
Network-Oriented Arbitration

- When multiple microprocessors share a bus (sometimes called **network-on-chip, NoC**)
  - Arbitration typically built into bus protocol
  - Separate processors may try to write simultaneously causing collisions
    - Data must be resent
    - Don't want to start sending again at the same time
      - Statistical methods can be used to reduce chances
- Typically used for connecting multiple IPs
  - Trend – use to connect multiple on-chip processors

Multilevel Bus Architectures

- Single bus is not enough for all communication
  - Peripherals would need high-speed, processor-specific bus interface
    - Excess gates, power consumption, and cost; less portable
  - Too many peripherals slows down bus
- Processor-local bus
  - High speed, wide, most frequent communication
  - Connects microprocessor, cache, memory controllers, etc.
- Peripheral bus
  - Lower speed, narrower, less frequent communication
  - Typically industry standard bus (ISA, PCI) for portability
- Bridge
  - Single-purpose processor converts communication between busses

Multilevel Bus Architectures

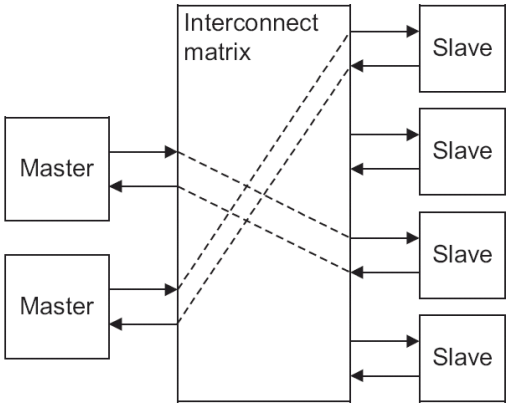


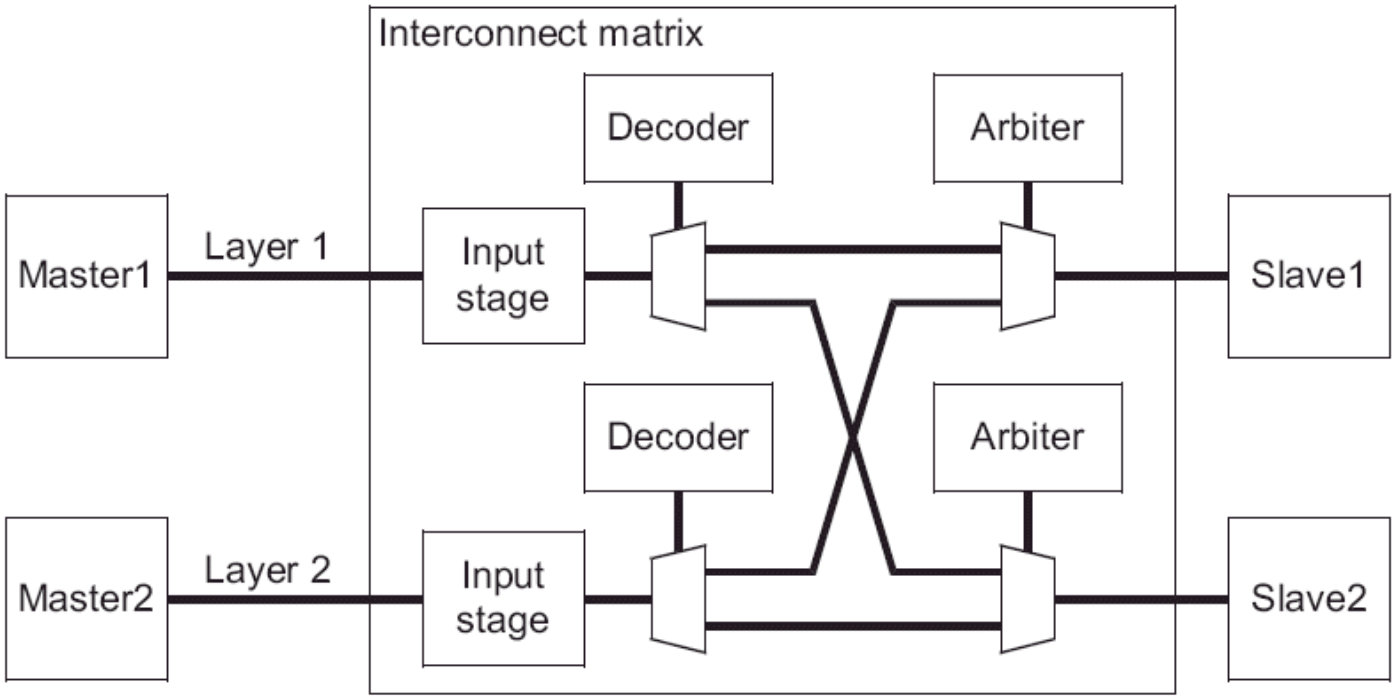
Multi-Layer AHB

- Multi-layer AHB
  - Enables parallel access paths between multiple masters and slaves by an interconnection matrix (bus matrix)
  - Increase the overall bus bandwidth
  - More flexible system architecture
    - Make slaves local to a particular layer
    - Make multiple slaves appear as a single slave to the interconnection matrix
    - Multiple masters on a single layer

Multi-Layer AHB

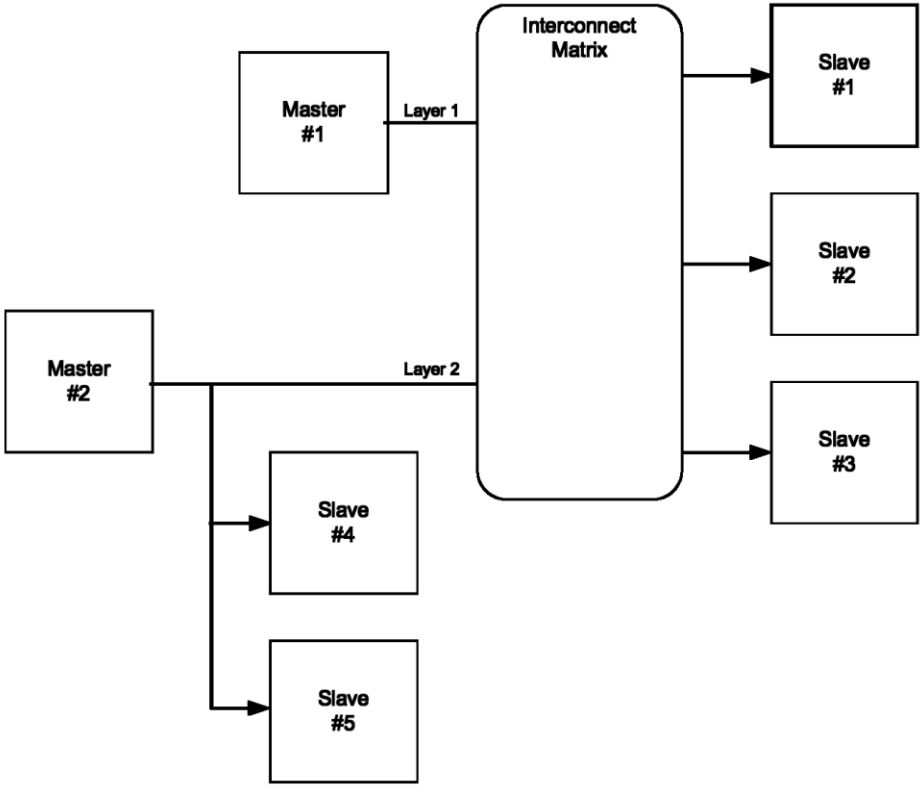
- A simple multi-layer system





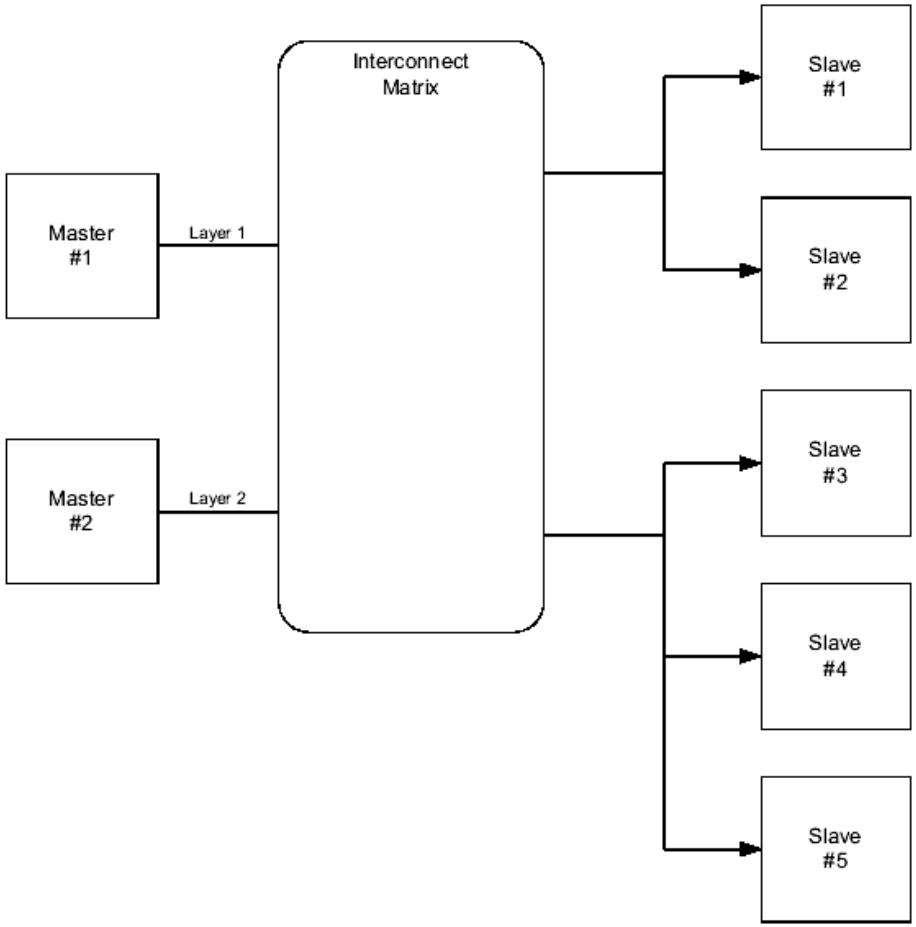
Multi-Layer AHB

- Local slaves
  - Slave #4 and Slave #5 can only be accessed by Master #2



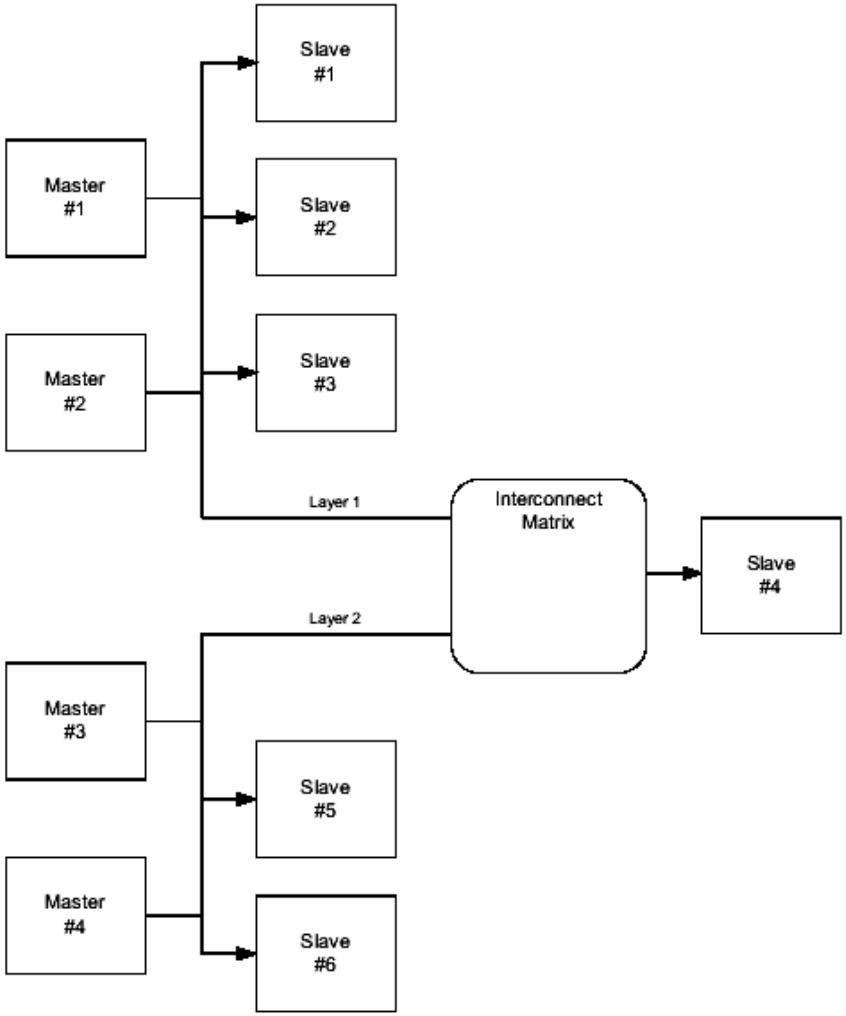
Multi-Layer AHB

- Multiple slaves on one slave port
  - Combine low-bandwidth slaves together
  - Combine salves usually accessed by the same master together



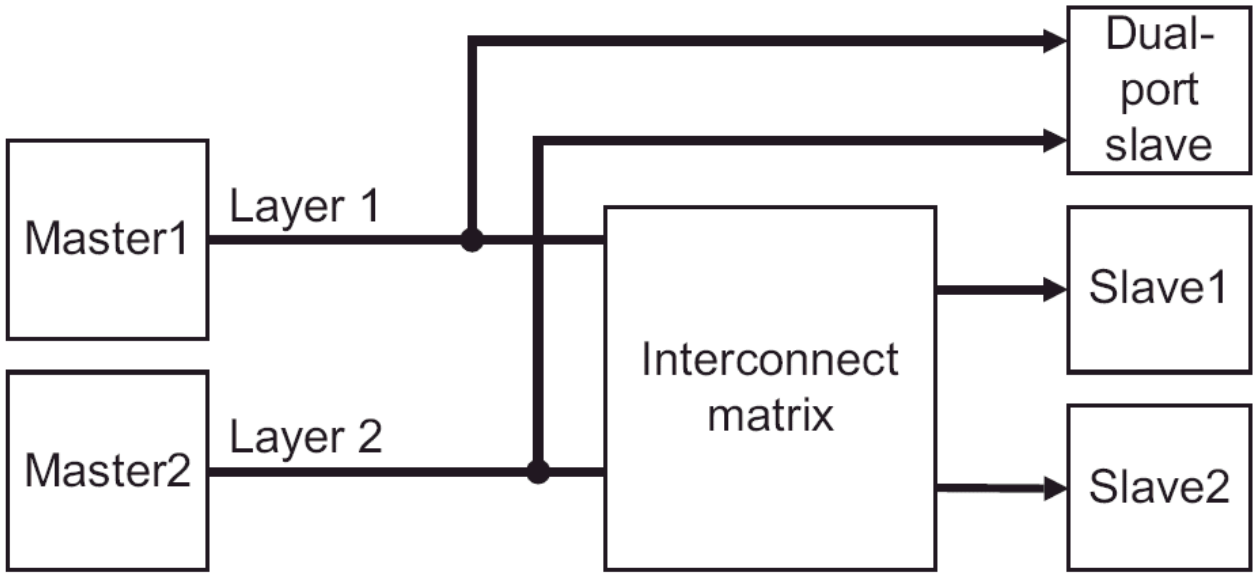
Multi-Layer AHB

- Multiple masters on one layer
  - Combine masters which have low-bandwidth requirements together
  - Combine special masters together

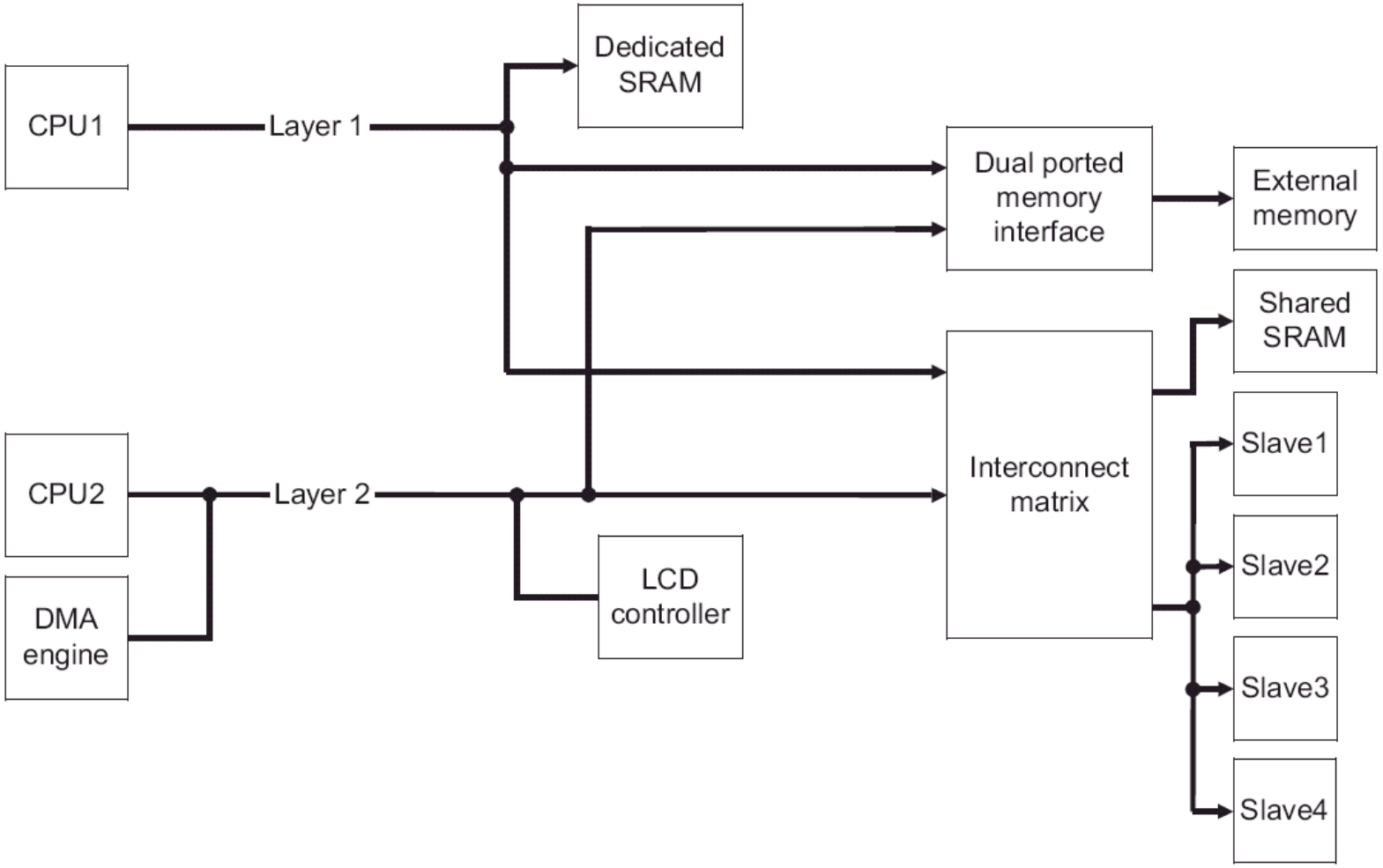


Multi-Layer AHB

- Multi-port slaves



Example


CPU1
Layer 1
Dedicated SRAM
Dual ported memory interface
External memory
Shared SRAM
Interconnect matrix
Slave1
Slave2
Slave3
Slave4
CPU2
DMA engine
Layer 2
LCD controller