Interfacing

Introduction

Embedded system functionality aspects

- ❖ Processing

  Transformation of data

  Implemented using processors

- ❖ Storage

  Retention of data

  Implemented using memory

- ❖ Communication

  Transfer of data between processors and memories

  Implemented using buses Called *interfacing*

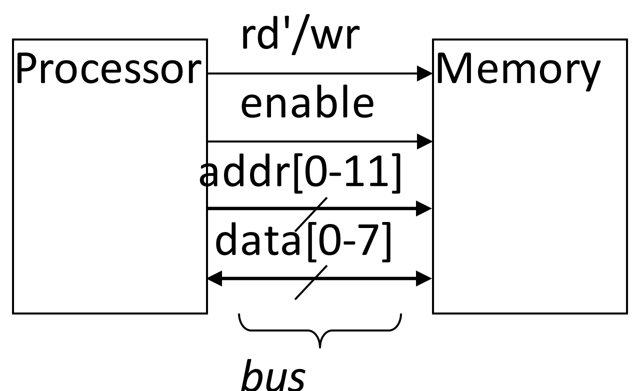
A simple bus

Wires:

- Uni-directional or bi-directional
- One line may represent multiple wires

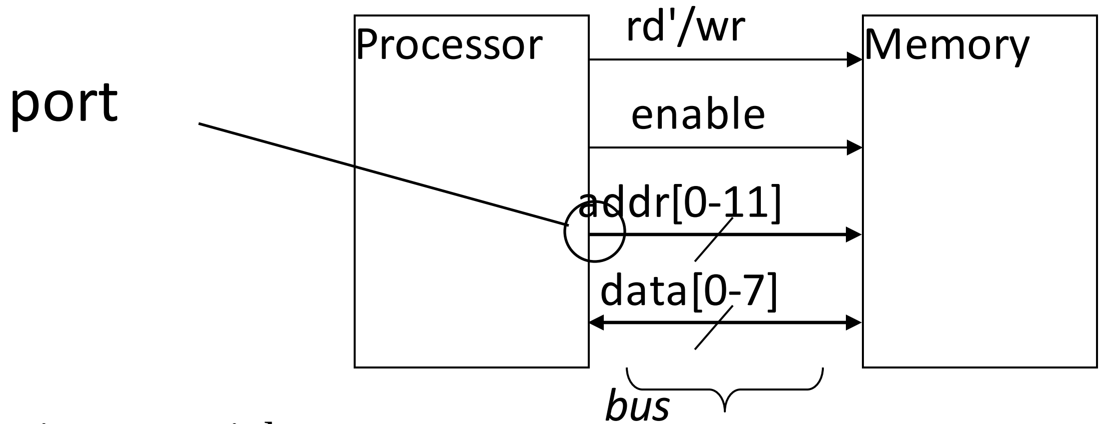
Bus

- 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

Ports



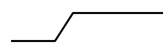
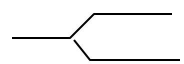
- Conducting device on periphery

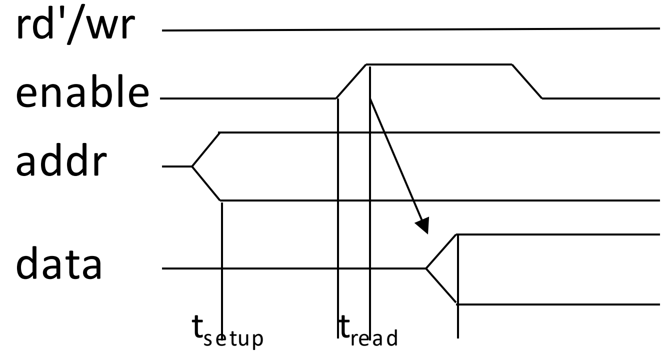
- Connects bus to processor or memory

- Often referred to as a *pin*
  - Actual pins on periphery of IC package that plug into socket on printed-circuit board
  - Sometimes metallic balls instead of pins
  - Today, metal “pads” connecting processors and memories within single IC

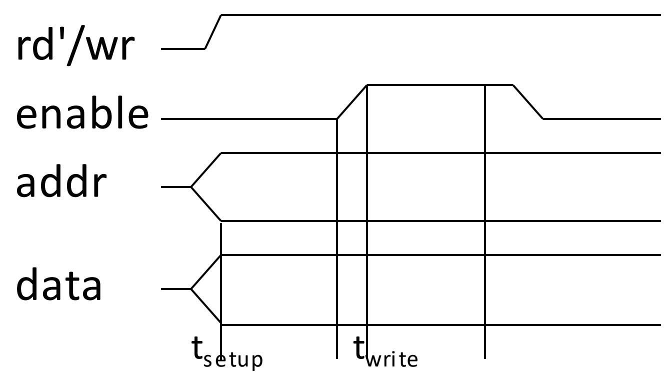
- Single wire or set of wires with single function
  - E.g., 12-wire address port

Timing Diagrams

- Most common method for describing a communication protocol.
- Time proceeds to the right on x-axis
- Control signal: low or high
  
  - May be active low (e.g., go', /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go' means go=0
- Data signal: not valid or valid
  
- Protocol may have subprotocols
  - Called bus cycle, e.g., read and write
  - Each may be several clock cycles
- Read example
  - *rd'/wr* set low,address placed on *addr* for at least t_{setup} time before *enable* asserted, enable triggers memory to place data on *data* wires by time t_{read}



read protocol

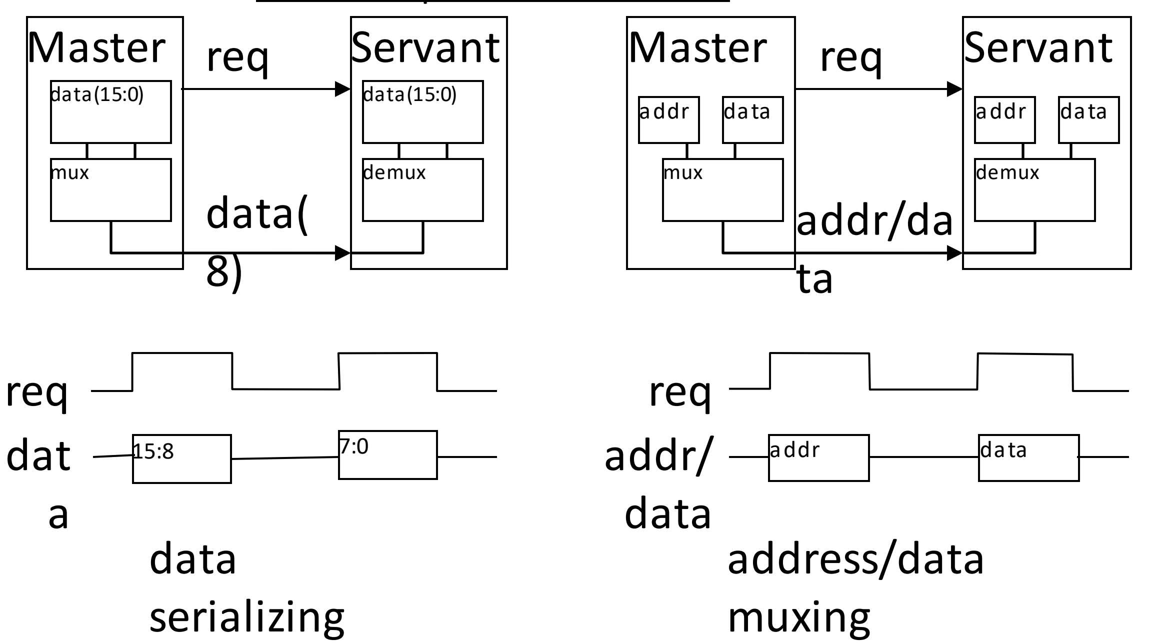


write protocol

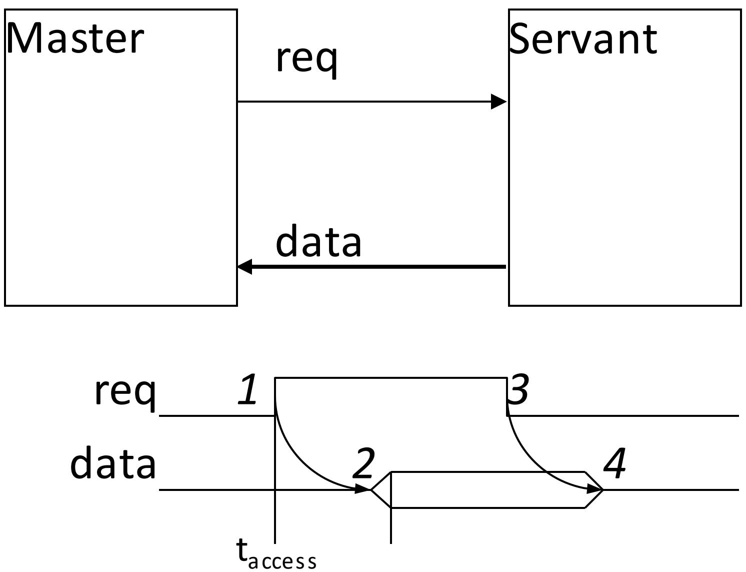
Basic protocol concepts

- 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

Time-multiplexed data transfer

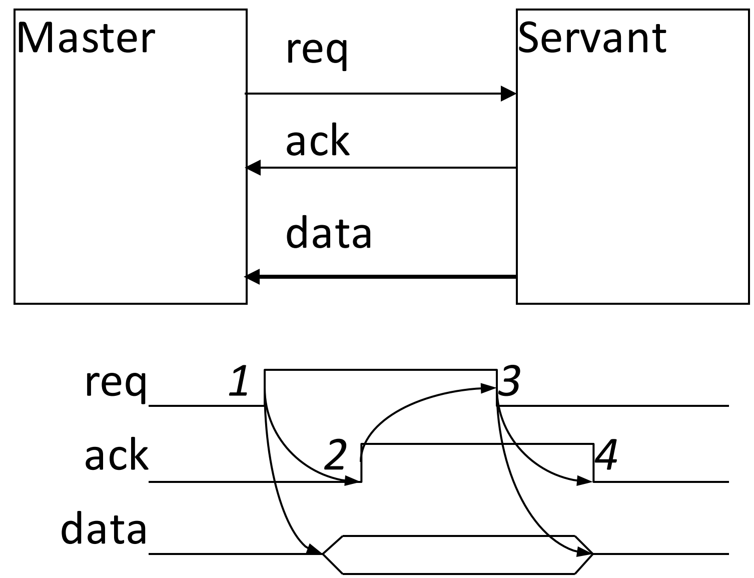


Basic protocol concepts: control methods



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

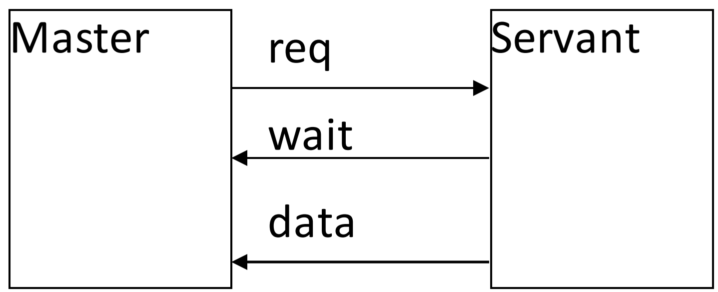
Strobe protocol

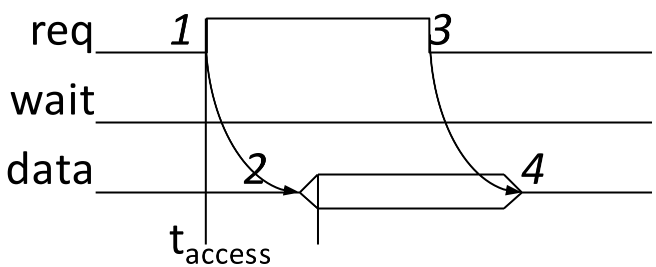


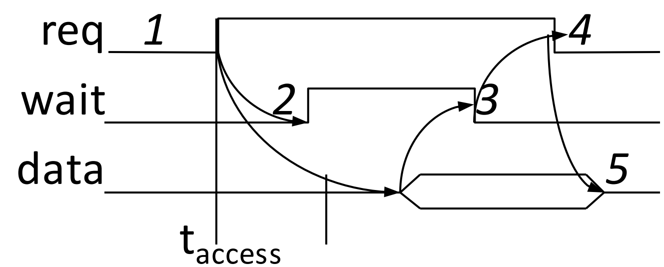
1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

Handshake protocol

A strobe/handshake compromise







1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}** (wait line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

1. Master asserts *req* to receive data
2. Servant can't put data within **t_{access}, asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

Fast-response case

Slow-response case

Microprocessor 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
- ❖ E.g., P0 = 0xFF; v = P1.2; -- P0 and P1 are 8-bit ports

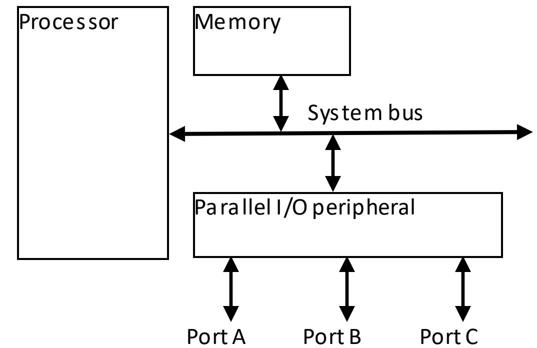
Bus-based I/O

- ❖ 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

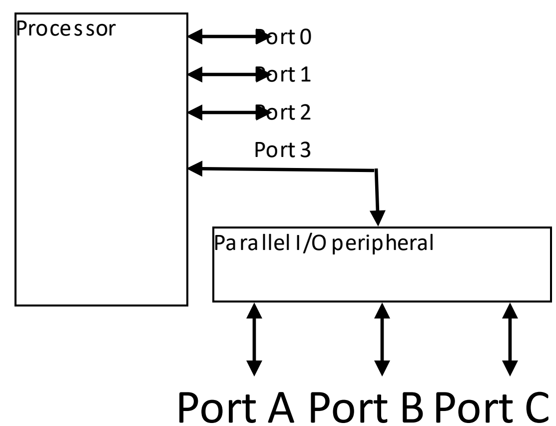
- ❖ When processor only supports bus-based I/O but parallel I/O needed
- ❖ Each port on peripheral connected to a register within peripheral that is read/written by the processor



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

Extended parallel I/O

- ❖ When processor supports port-based I/O but more ports needed
- ❖ One or more processor ports interface with parallel I/O peripheral extending total number of ports available for I/O
- ❖ e.g., extending 4 ports to 6 ports in figure



Extended parallel I/O

Types of bus-based I/O: memory-mapped I/O and standard I/O

Memory-Mapped I/O	I/O Mapped I/O
Memory mapped IO uses the same address space for both memory and IO device .	IO mapped IO uses two separate address spaces for memory and IO device.
Only 1 set of read and write instruction lines.	I/O read and I/O write lines for I/O transfer. Memory read and Memory write lines for memory transfer.
No separate instruction like IN, OUT and MOV. The instruction used to manipulate the memory can be used for I/O devices.	IN and OUT instructions deals with I/O transfer. MOV instruction deals with memory transfer.

Microprocessor 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!

Microprocessor 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

Time

1(a): µP is executing its main program.

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

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

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.

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

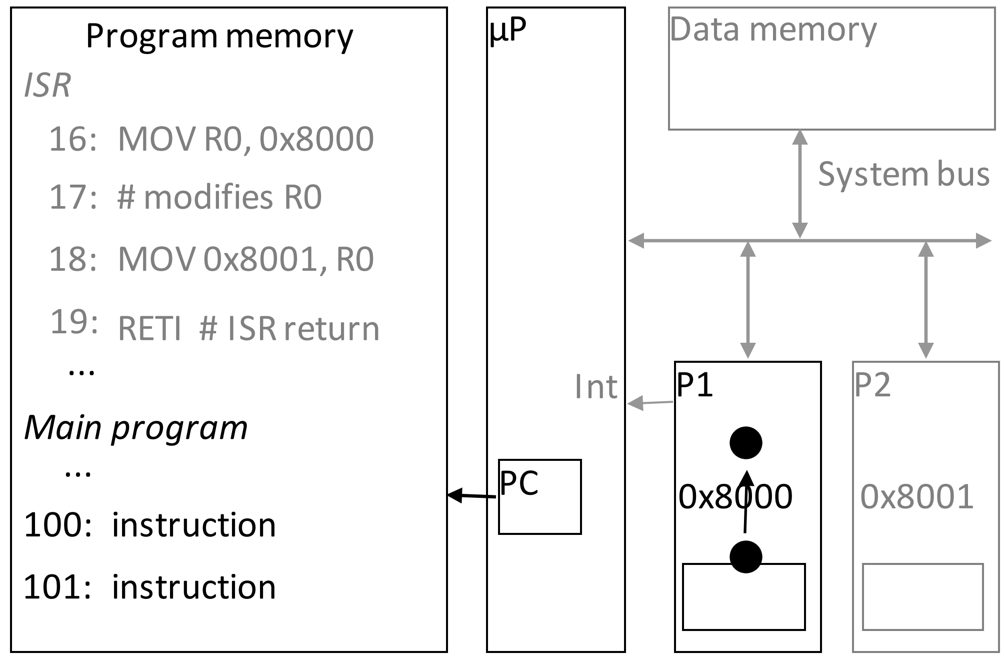
4(b): After being read, P1 de-asserts *Int*.

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

Interrupt-driven I/O using fixed ISR location

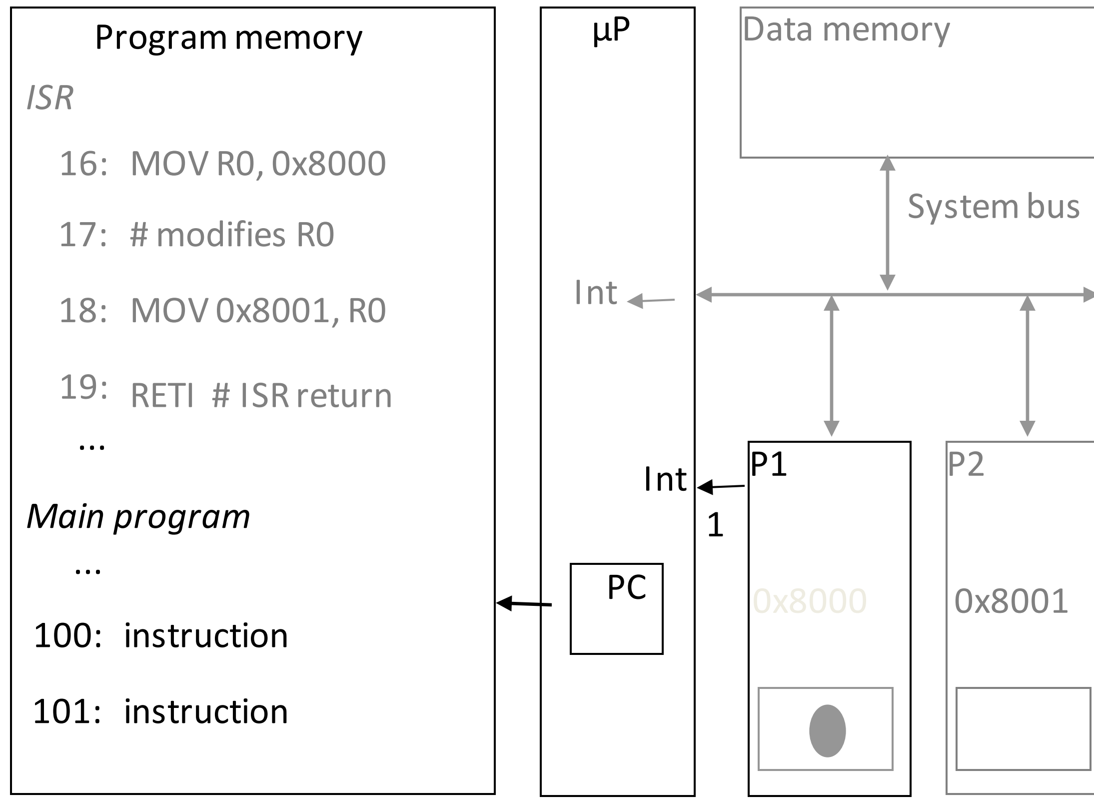
1(a): μP is executing its main program

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



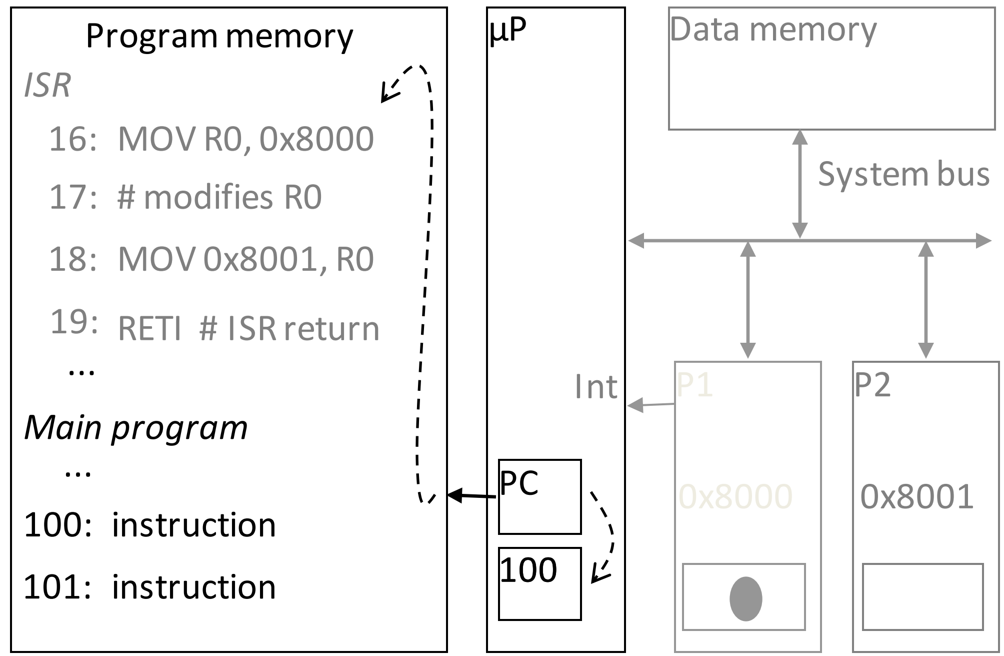
Interrupt-driven I/O using fixed ISR location

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



Interrupt-driven I/O using fixed ISR location

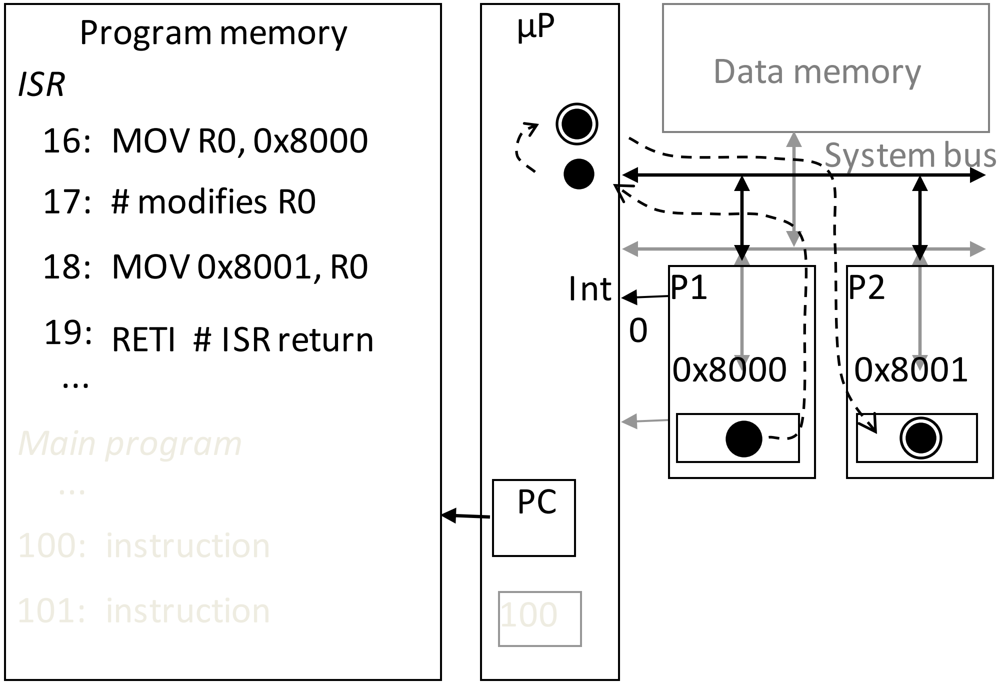
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-driven I/O using fixed ISR location

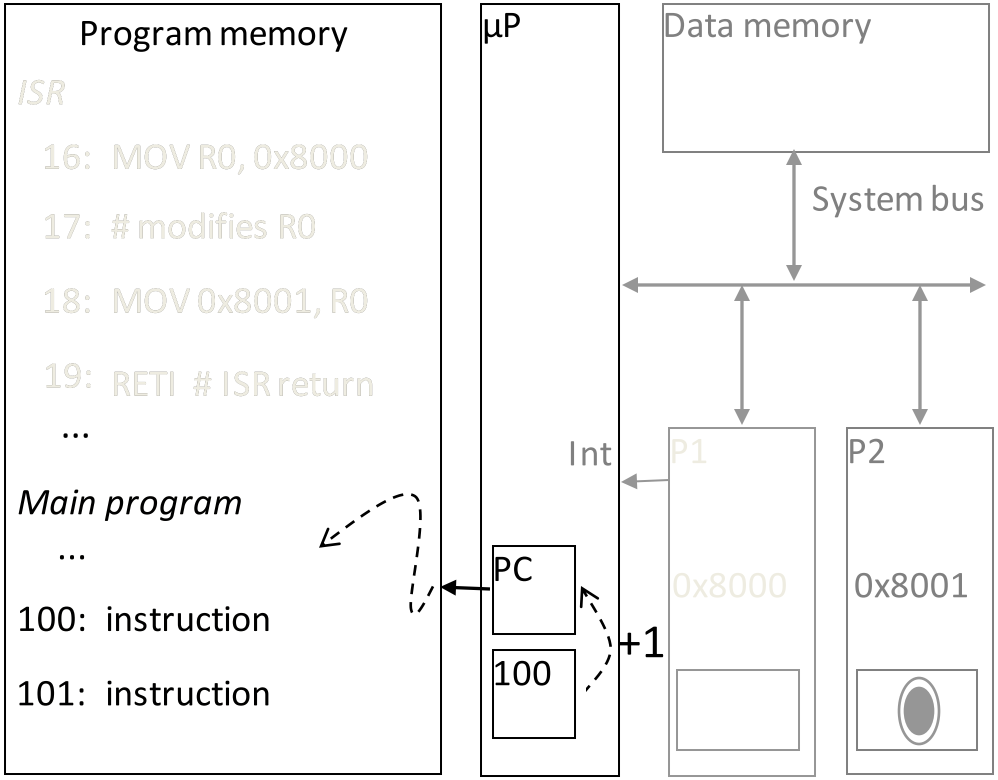
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

Time

1(a): µP is executing its main program.

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

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

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

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

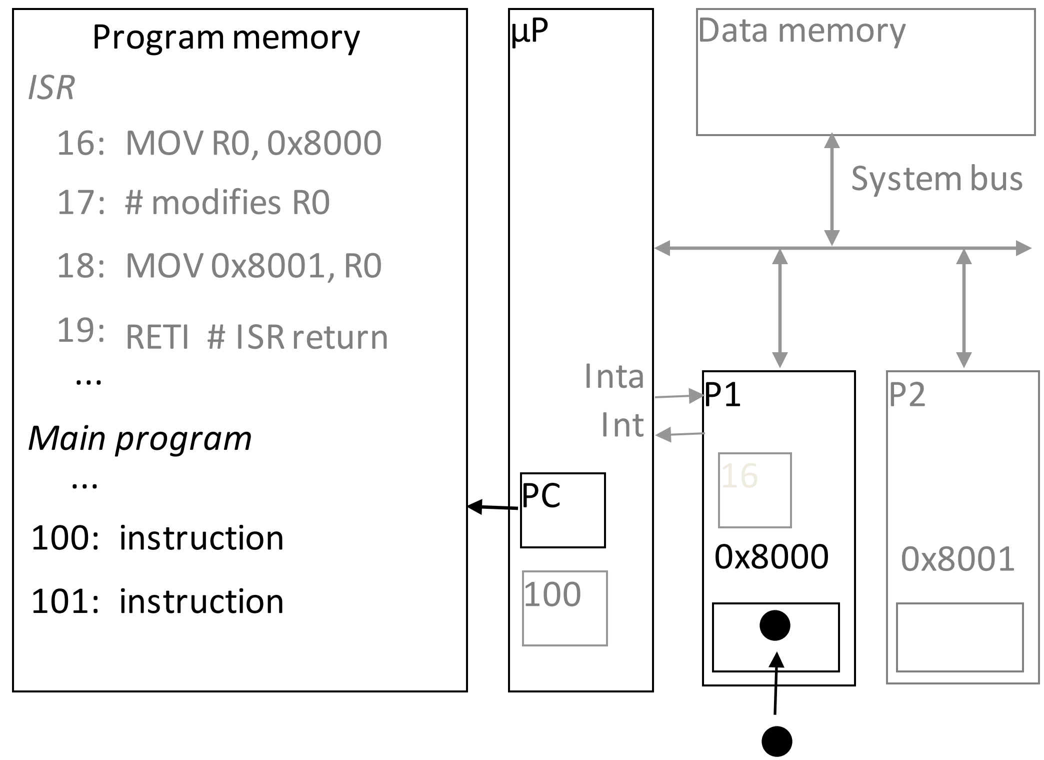
5(a): **µP jumps to the address on the bus (16)**. The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

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

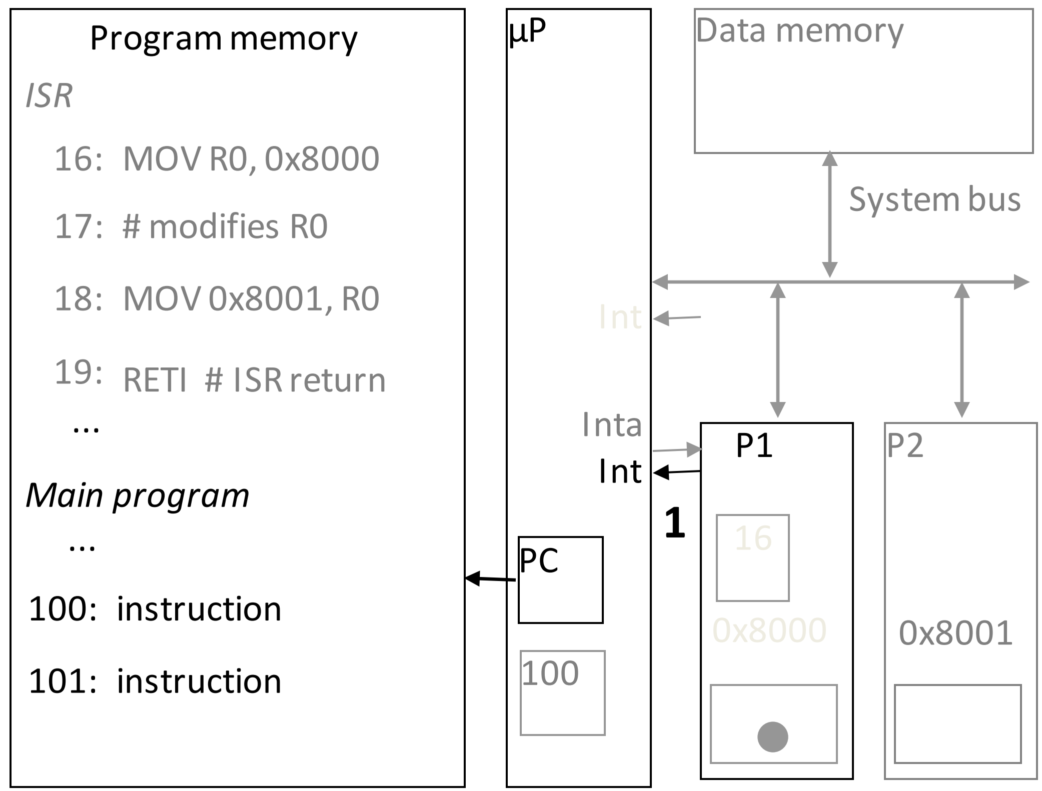
Interrupt-driven I/O using vectored interrupt

1(a): P is executing its main program
1(b): P1 receives input data in a register with address 0x8000.



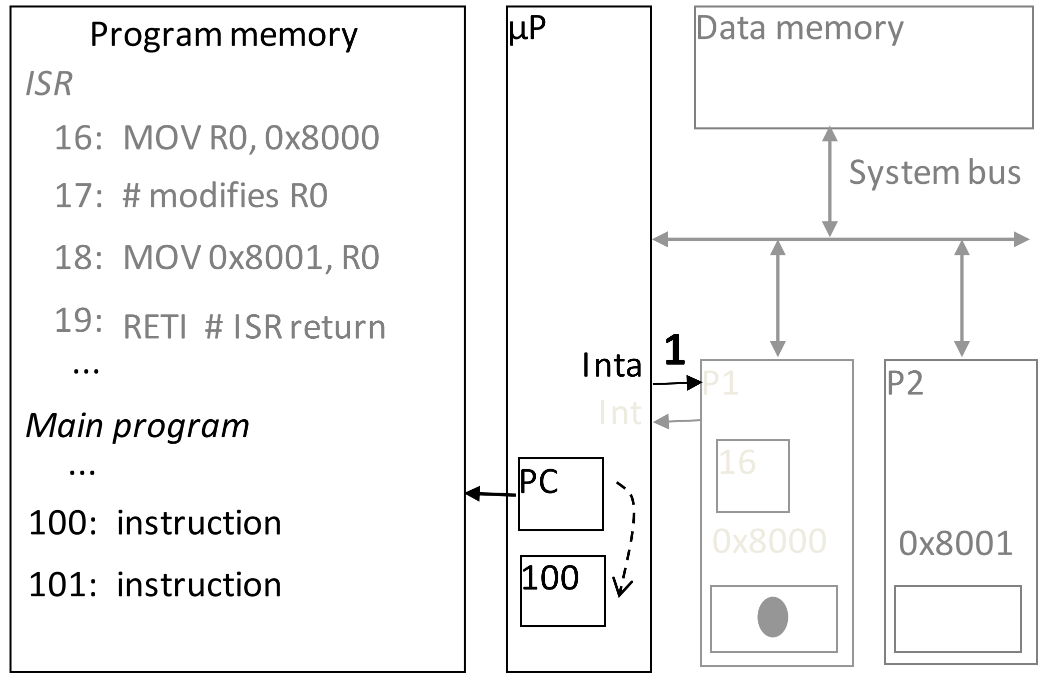
Interrupt-driven I/O using vectored interrupt

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



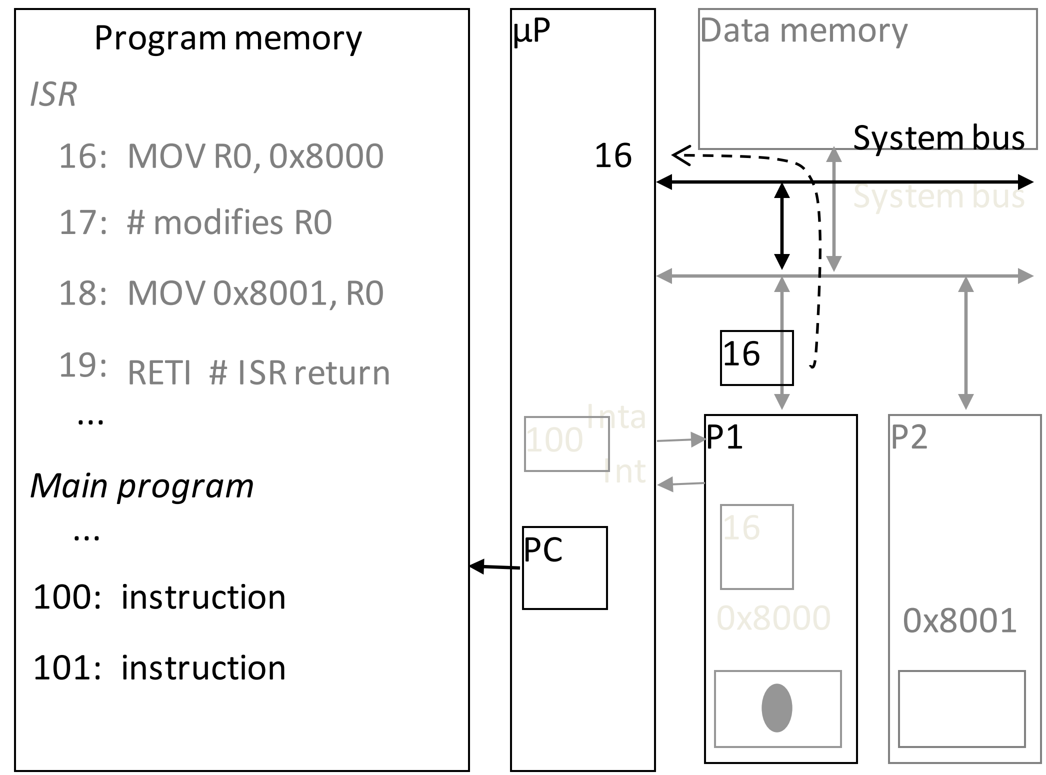
Interrupt-driven I/O using vectored interrupt

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

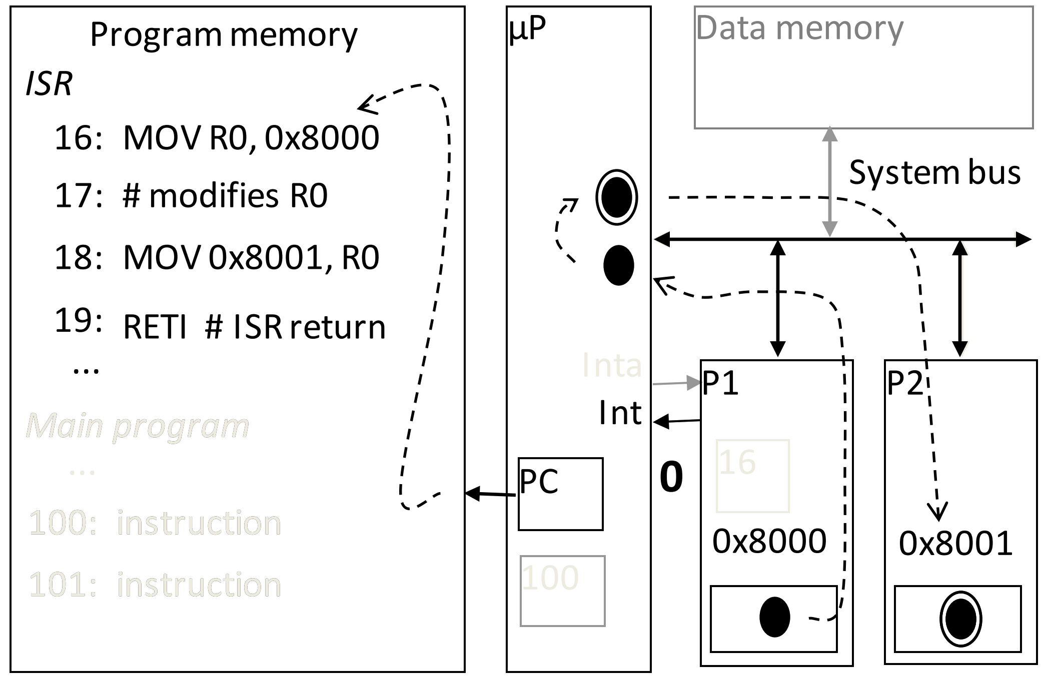
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-driven I/O using vectored interrupt

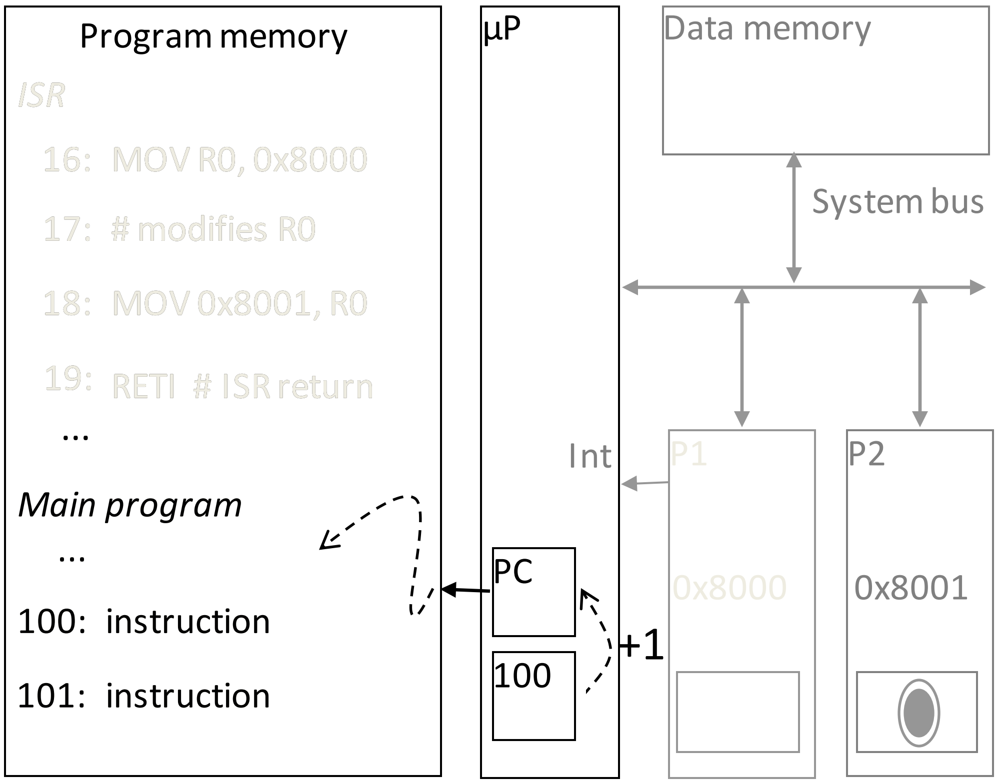
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

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

Jump to ISR

- ❖ Some microprocessors treat jump 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

Direct memory access

- 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 inefficient
  - Regular program must wait
- DMA controller more efficient
  - Separate single-purpose processor
  - Microprocessor relinquishes control of system bus to DMA controller
  - Microprocessor can meanwhile execute its regular program
    - No inefficient storing and restoring state due to ISR call
    - Regular program need not 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

Peripheral to memory transfer *without* DMA, using vectored interrupt

Time

1(a): µP is executing its main program.

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

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

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

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

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.

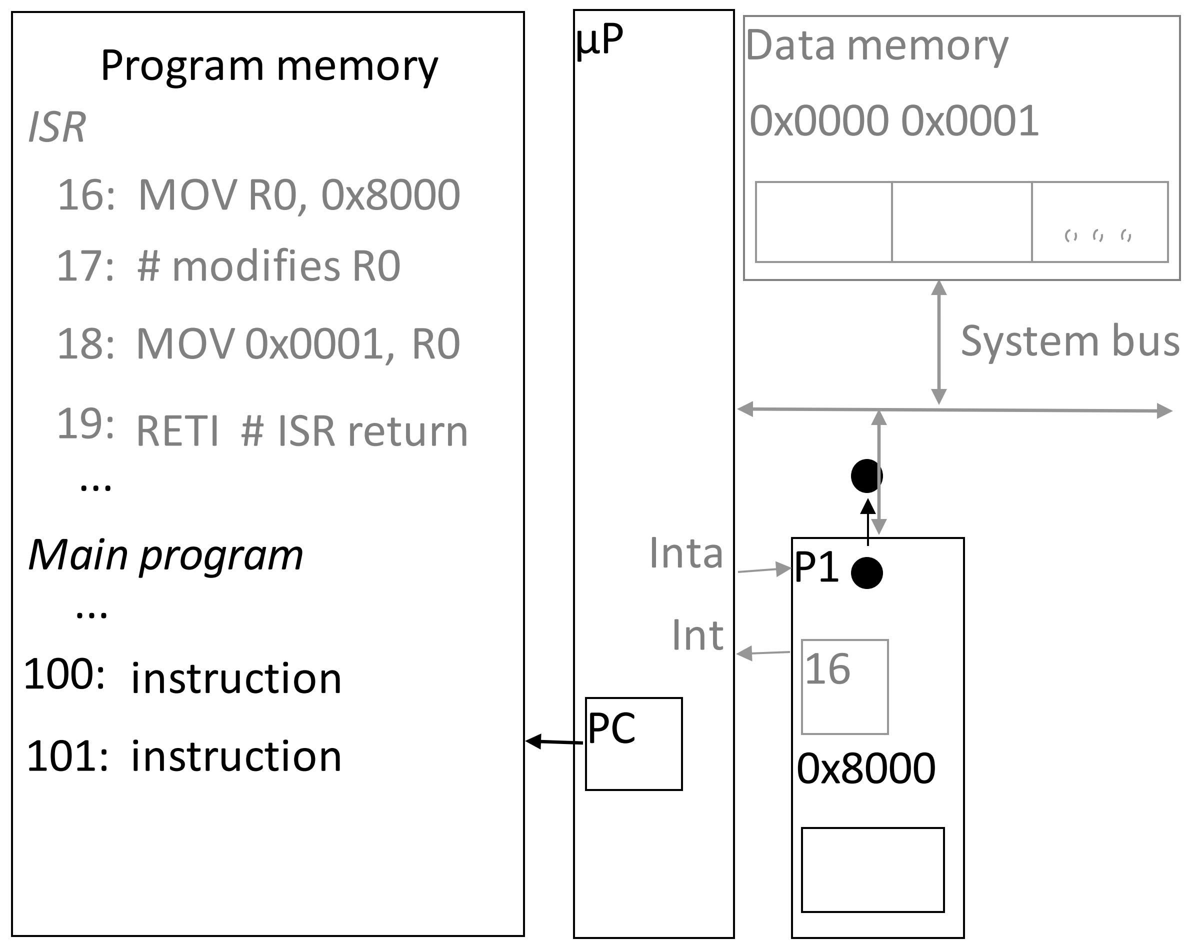
5(b): After being read, P1 deasserts *Int*.

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

Peripheral to memory transfer *without* DMA, using vectored interrupt

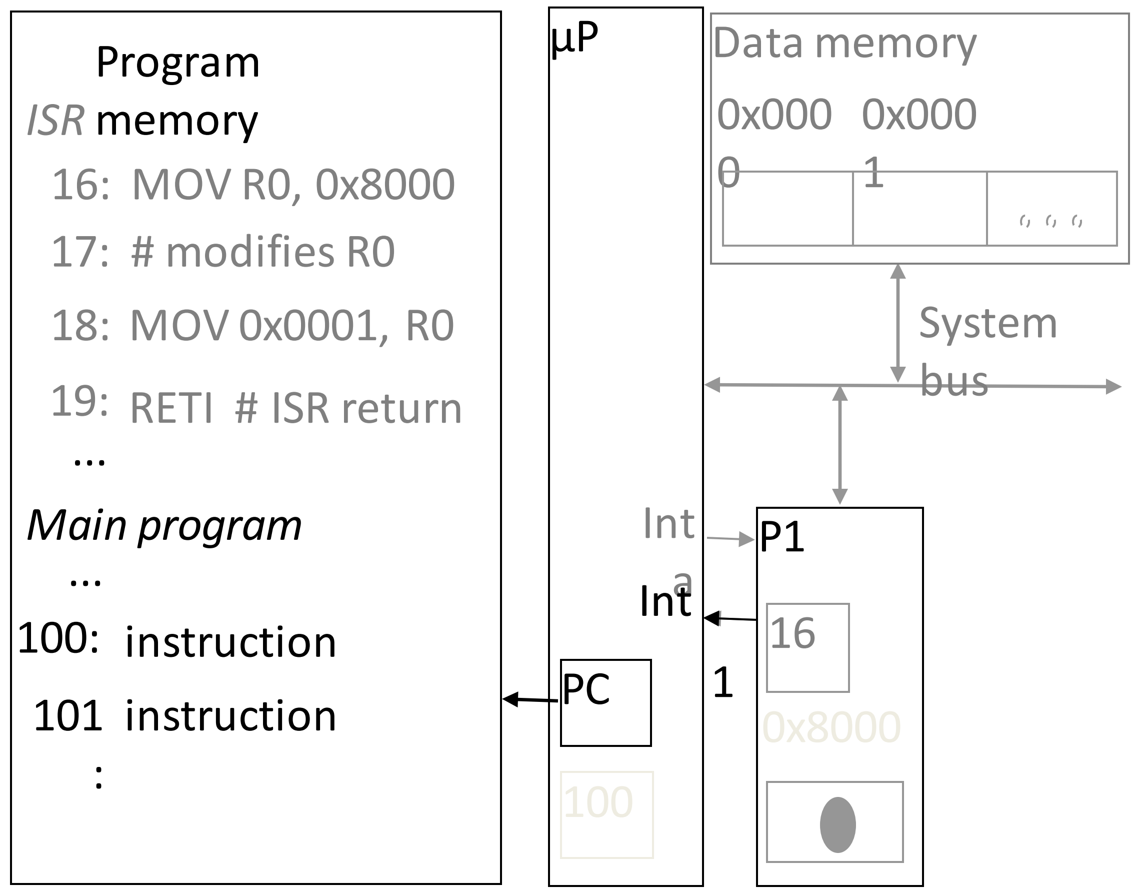
1(a): μP is executing its main program

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



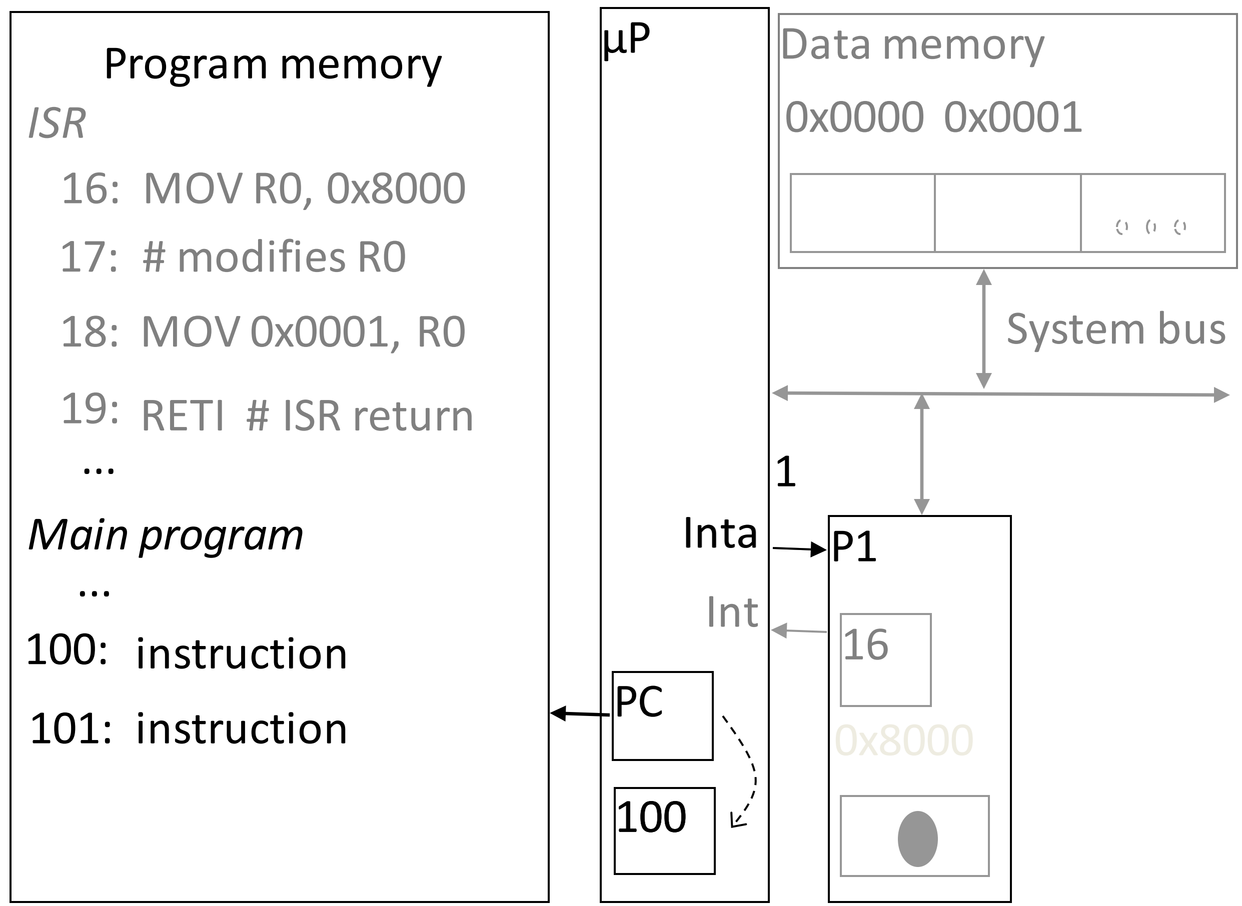
Peripheral to memory transfer *without* DMA, using vectored interrupt

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



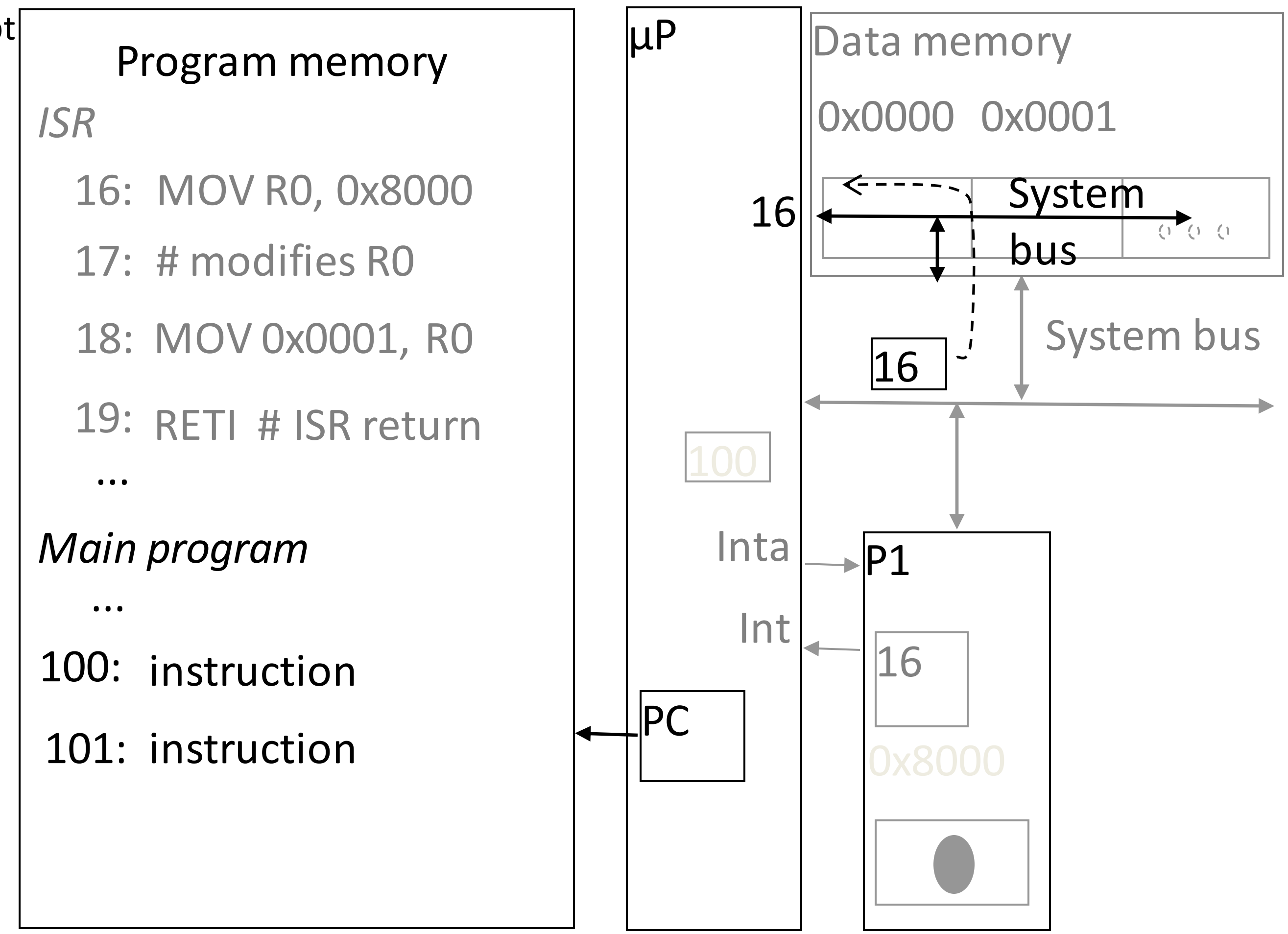
Peripheral to memory transfer *without* DMA, using vectored interrupt

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, using vectored interrupt (cont')

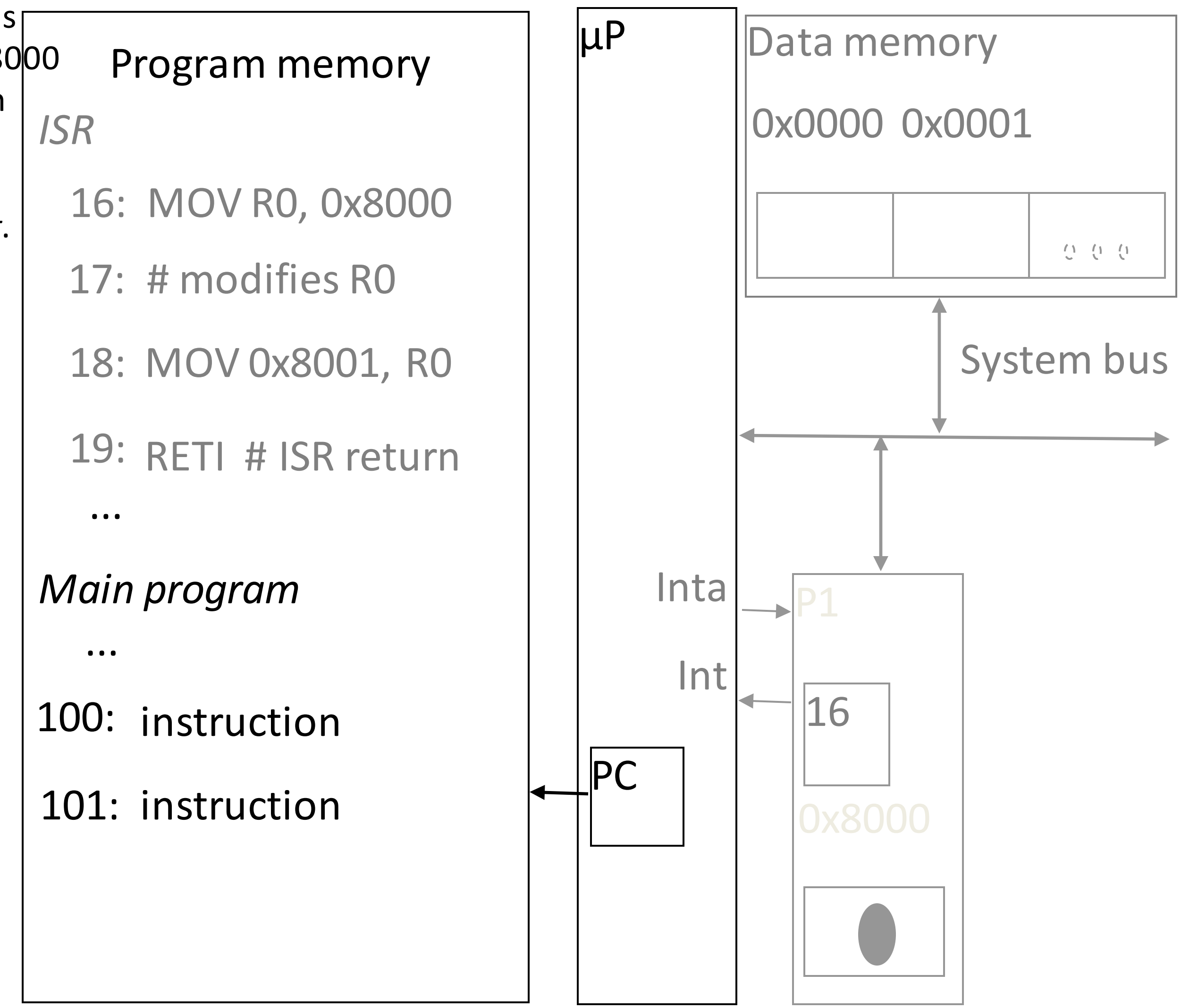
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

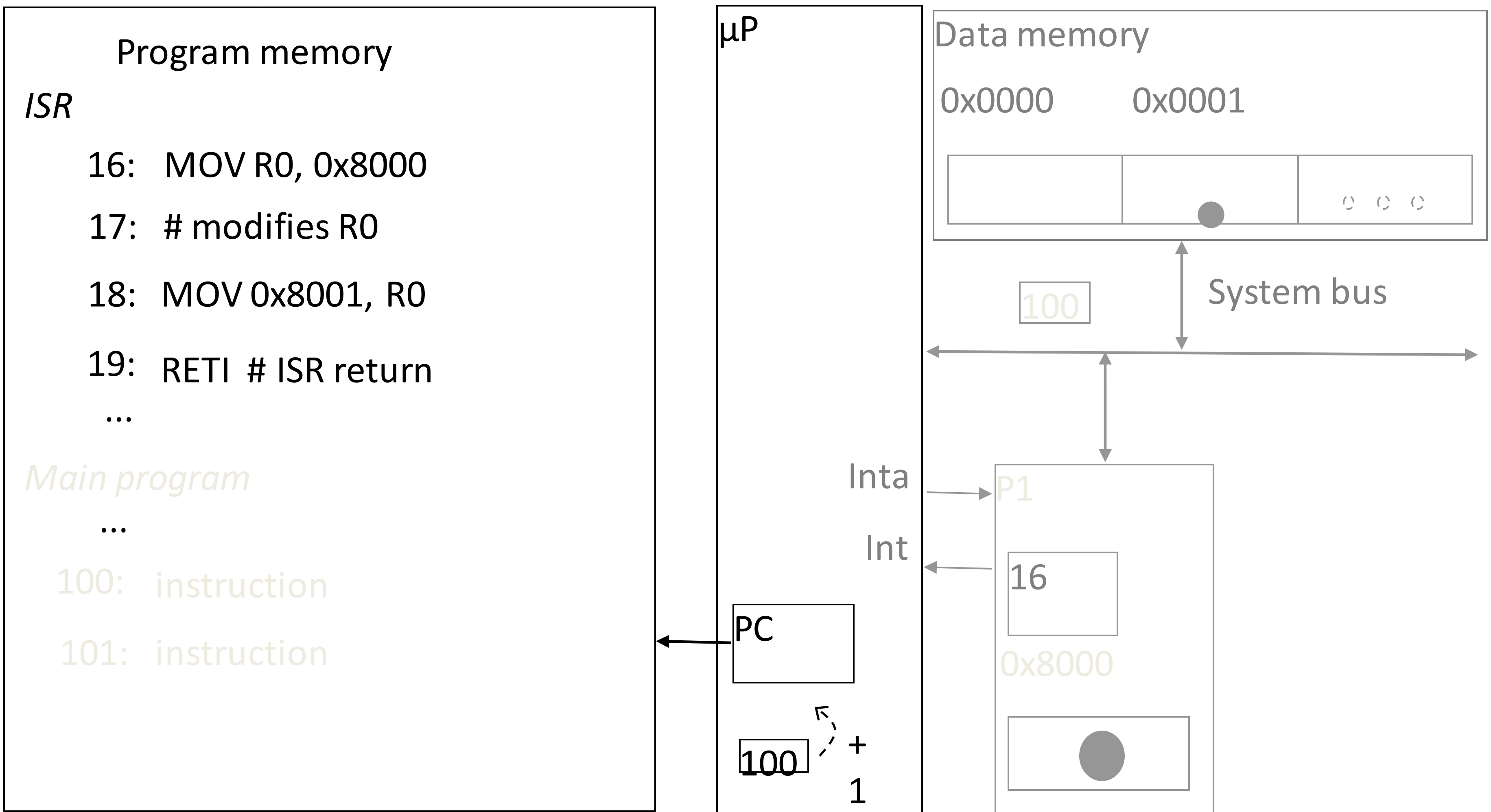
5(a): μP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.

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



Peripheral to memory transfer *without* DMA, using vectored interrupt (cont')

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



Peripheral to memory transfer with DMA

Time

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

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

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

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

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.

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

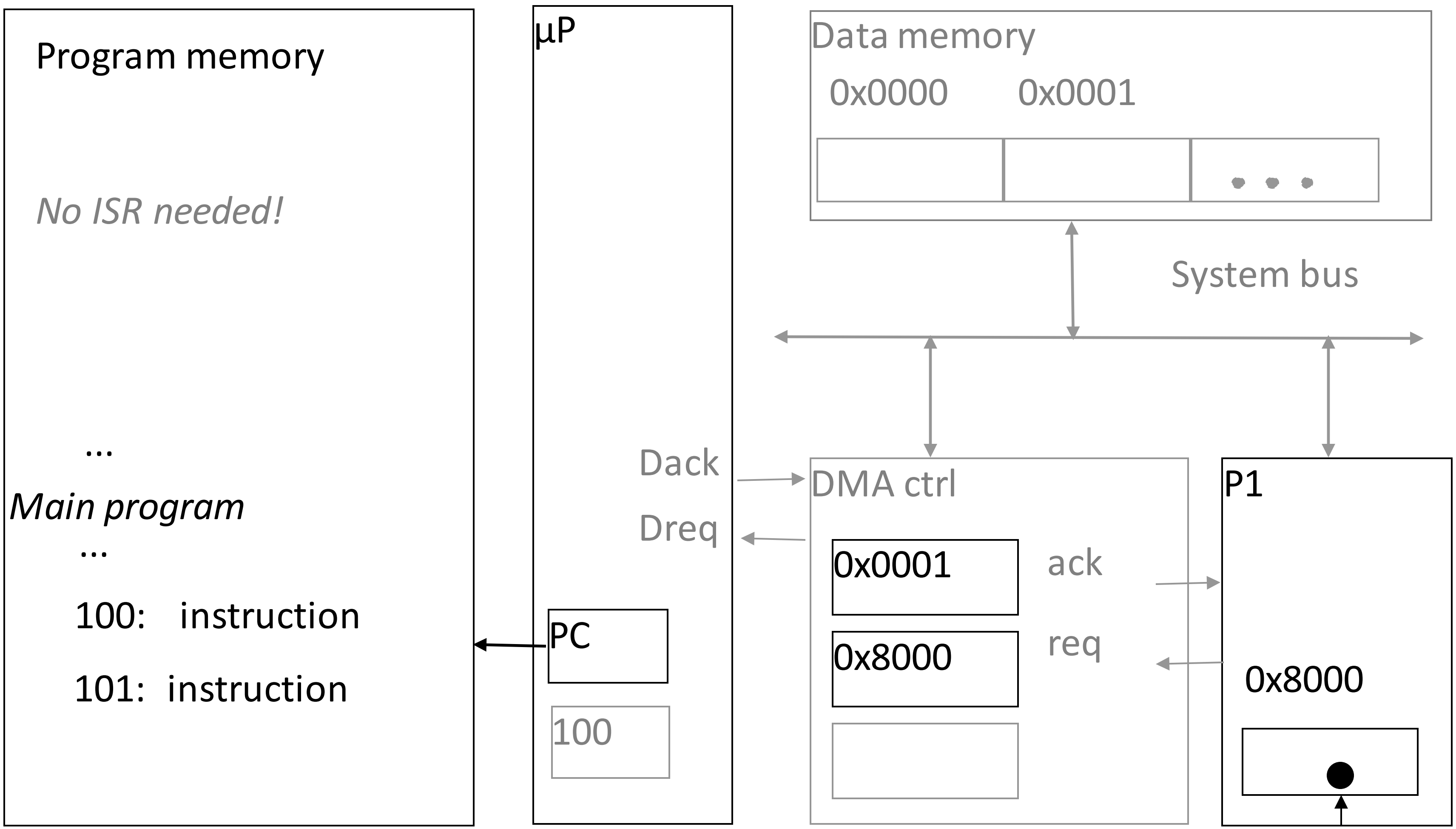
6:. DMA de-asserts *Dreq* and ack completing handshake with P1.

7(a): µP de-asserts *Dack* and resumes control of the bus.

7(b): P1 de-asserts *req*.

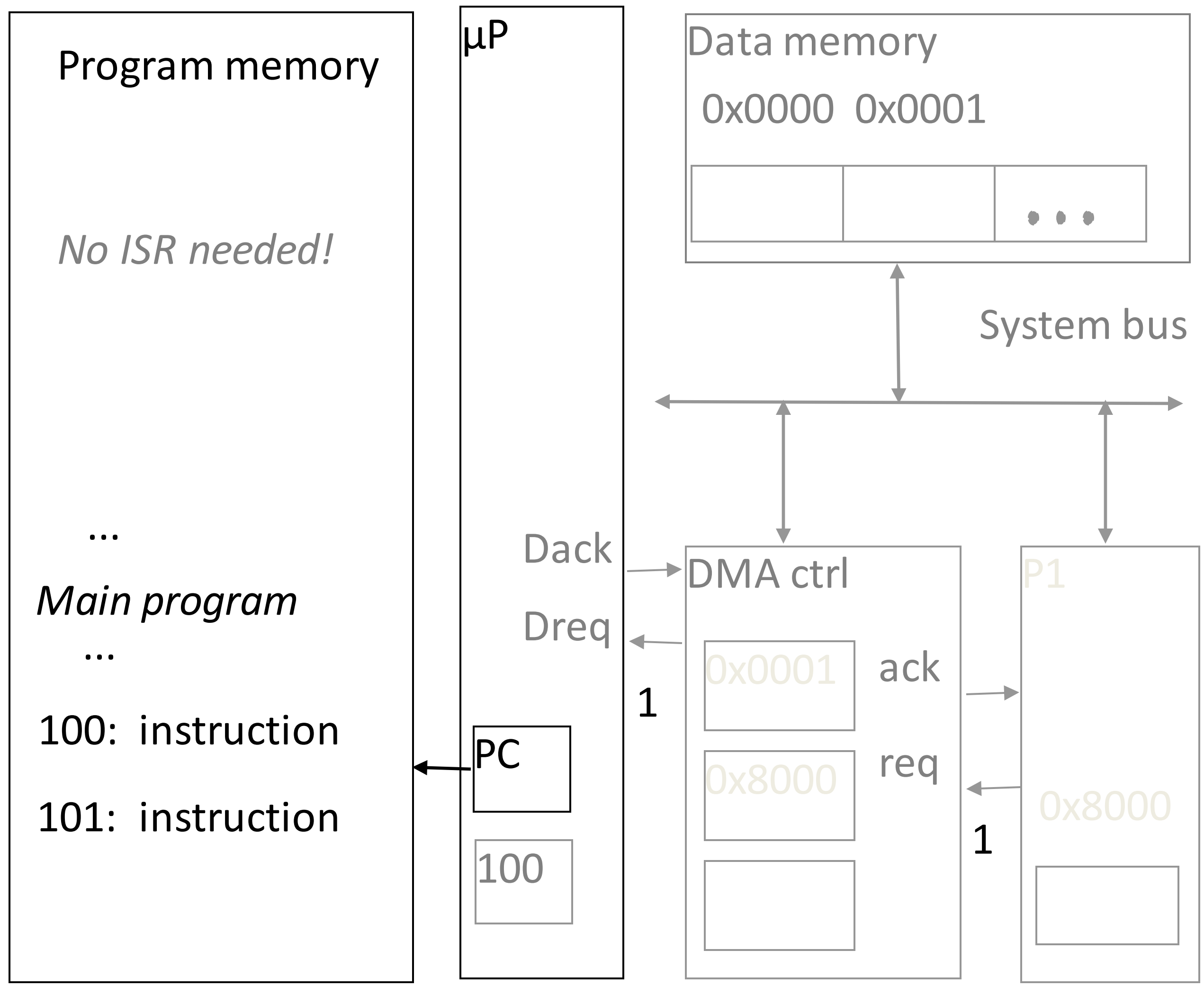
Peripheral to memory transfer with DMA

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.



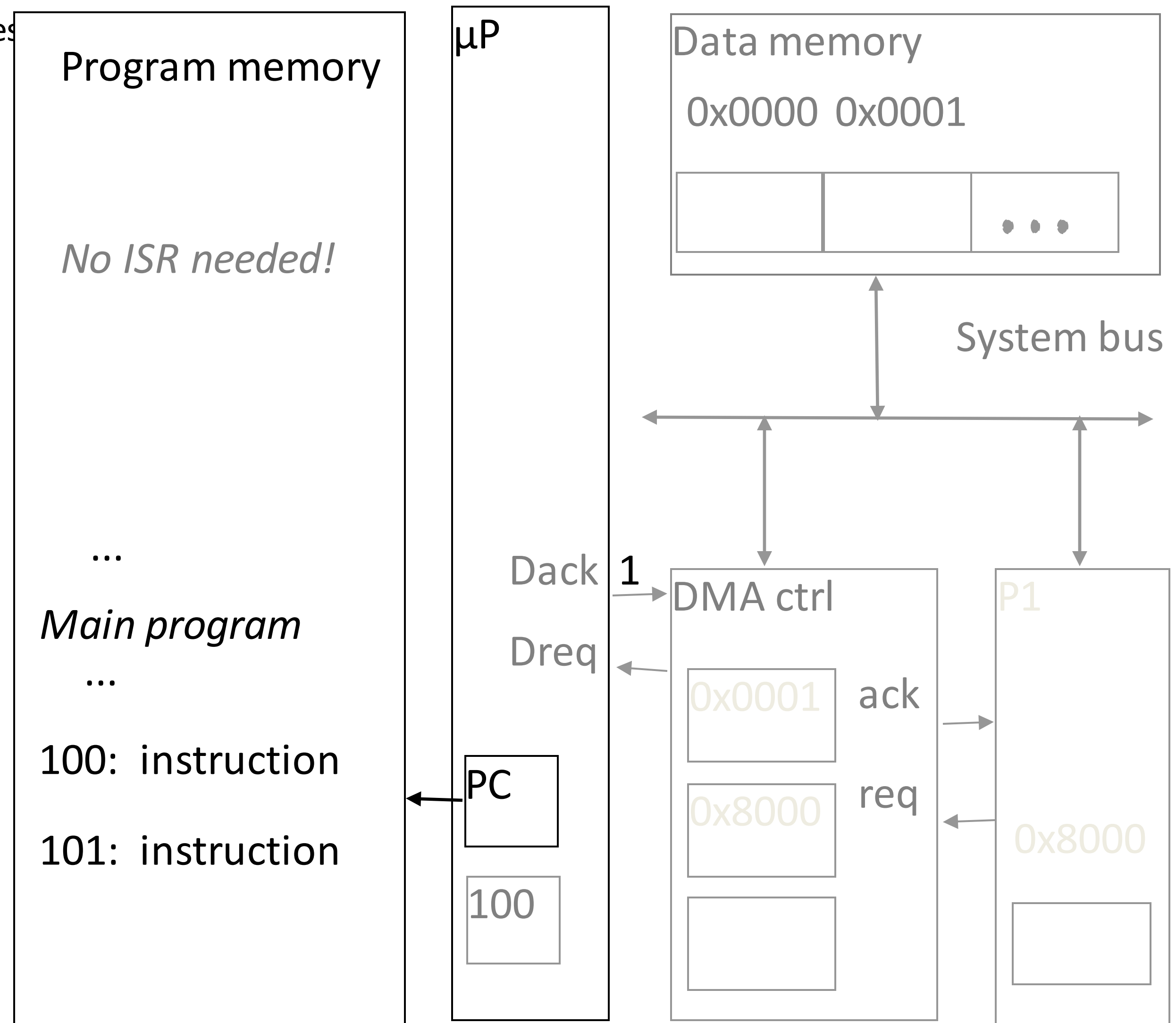
Peripheral to memory transfer with DMA (cont')

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 (cont')

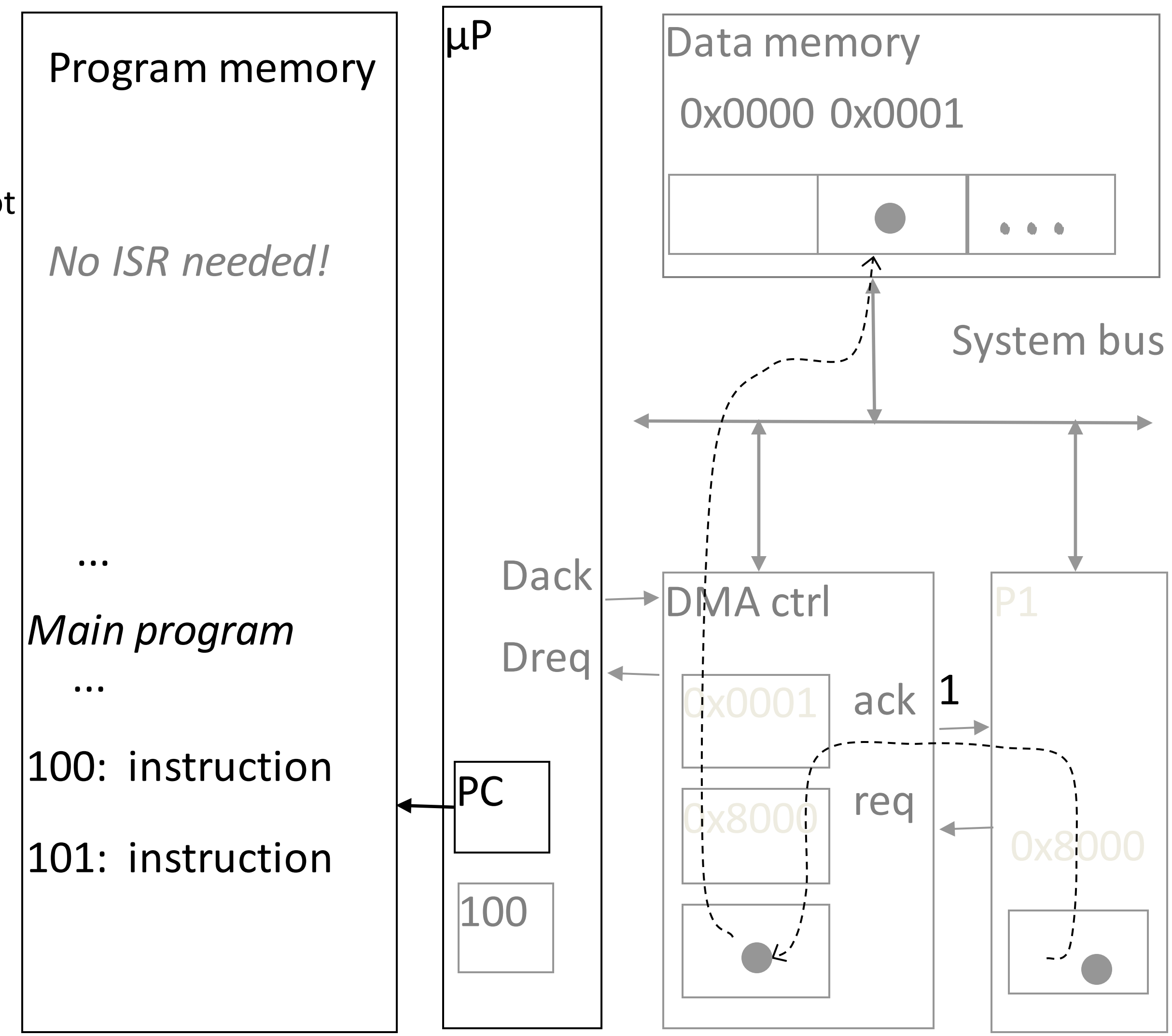
4: After executing instruction 100, μP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, μP stalls only if it needs the system bus to continue executing.



Peripheral to memory transfer with DMA (cont')

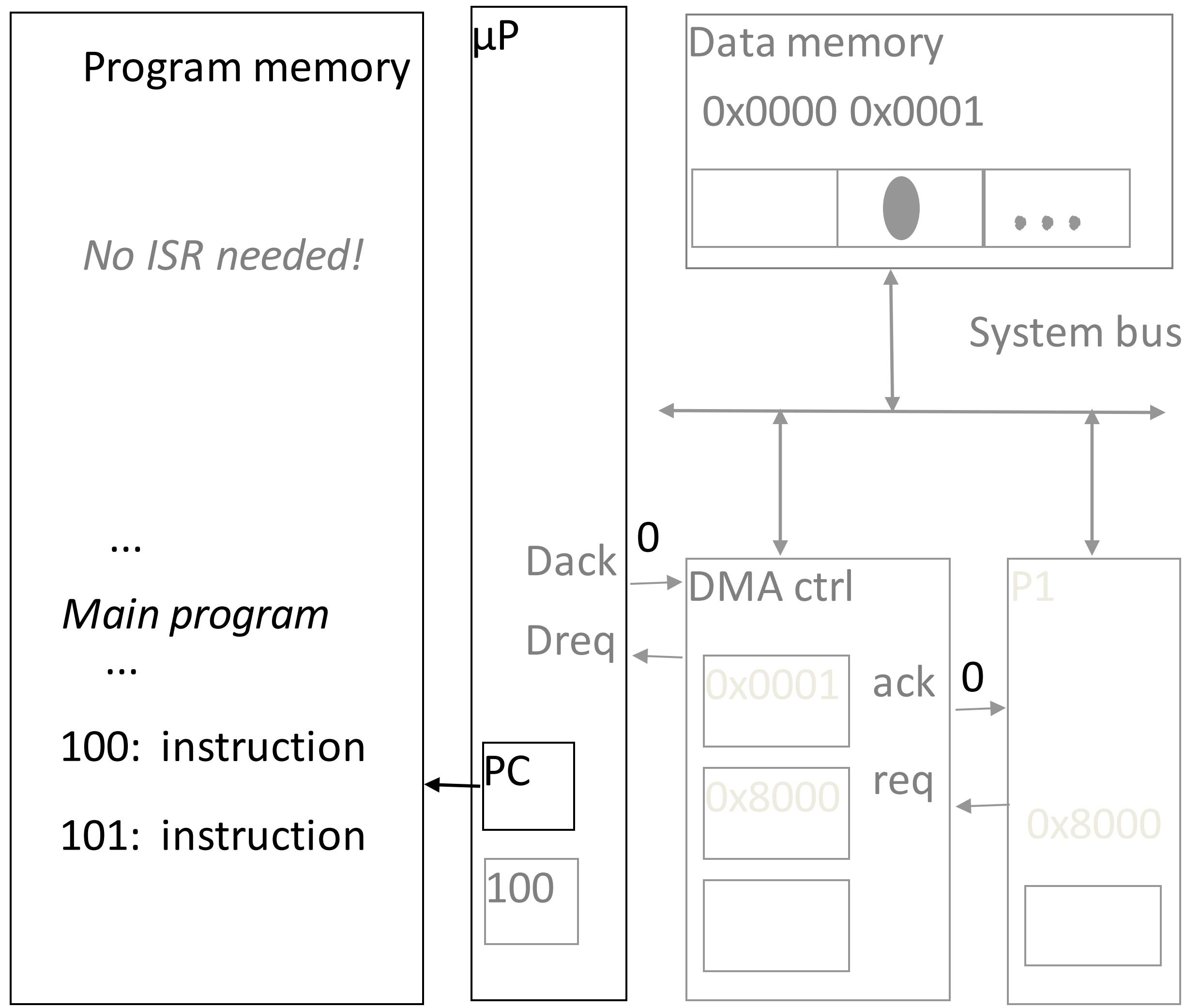
5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.

(Meanwhile, processor still executing if not stalled!)



Peripheral to memory transfer with DMA (cont')

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

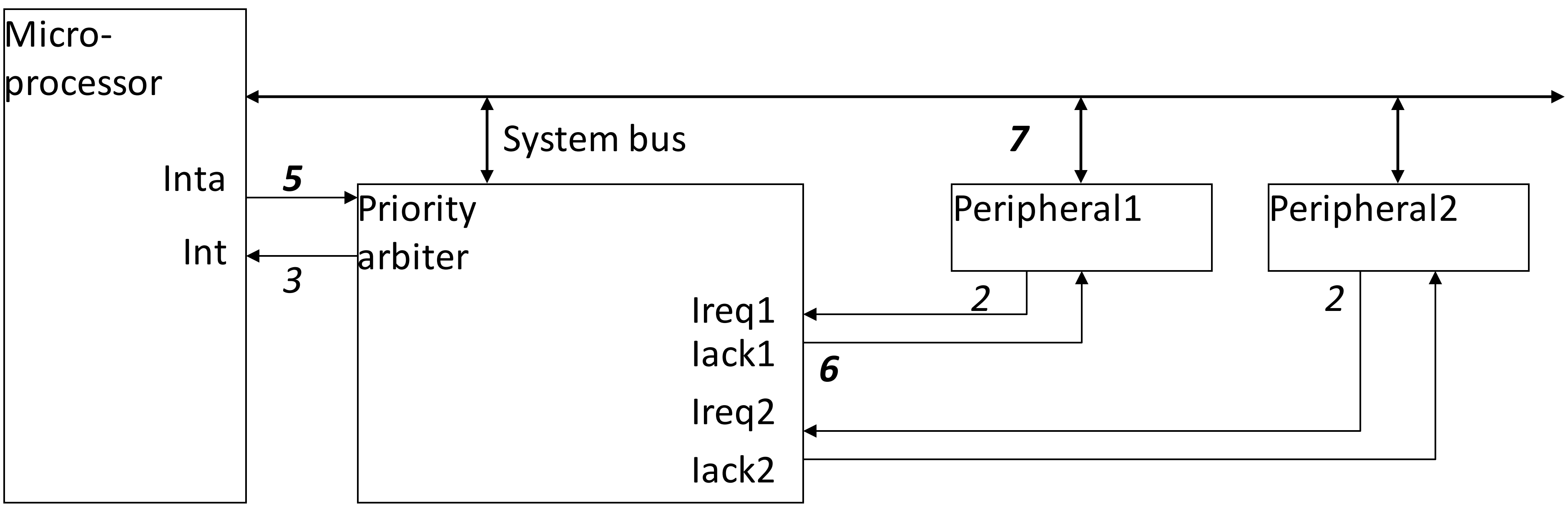


Arbitration: Priority arbiter

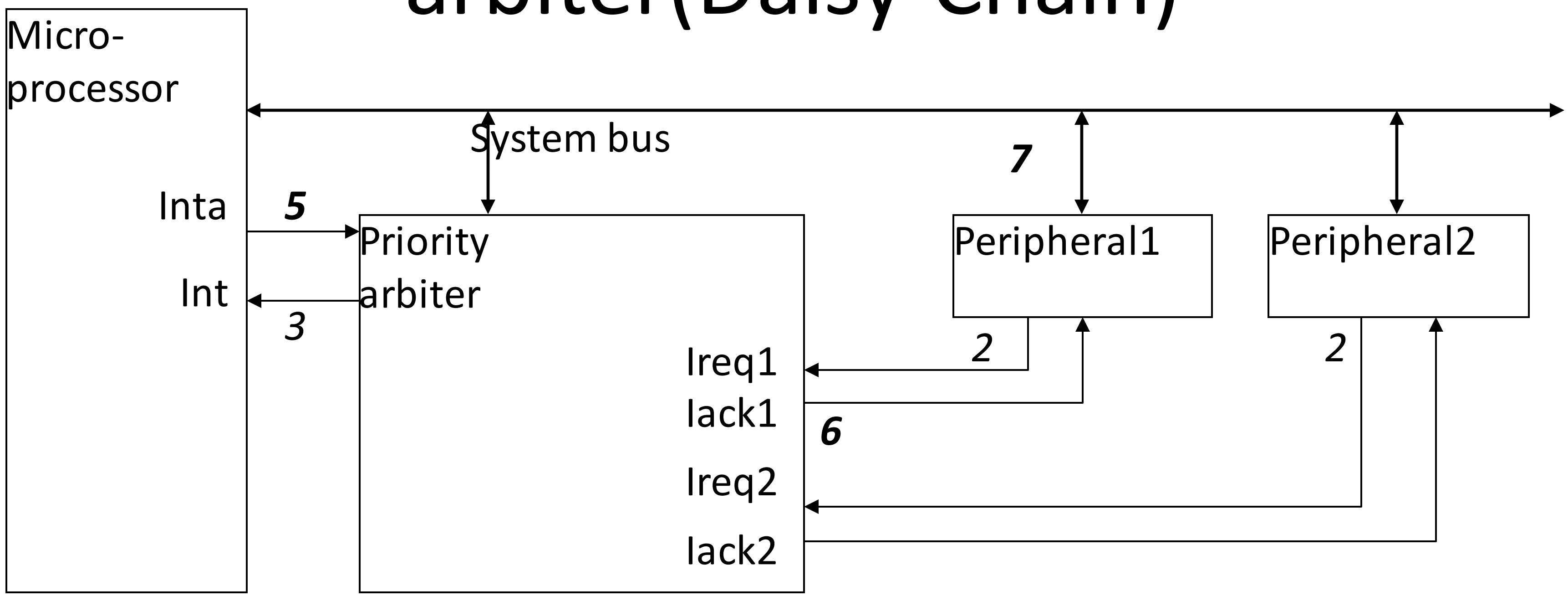
Consider the situation where multiple peripherals request service from single resource (e.g., microprocessor, DMA controller) simultaneously - which gets serviced first?

Priority arbiter

- Single-purpose processor
- Peripherals make requests to arbiter, arbiter makes requests to resource
- Arbiter connected to system bus for configuration only



Arbitration using a priority arbiter(Daisy Chain)



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor asserts *Inta*.
6. Priority arbiter asserts *Iack1* to acknowledge Peripheral1.
7. Peripheral1 puts its interrupt address vector on the system bus
8. Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).
9. Microprocessor resumes executing its program.

Arbitration: Priority arbiter

Types of priority

Fixed priority

- ❖each peripheral has unique rank
- ❖highest rank chosen first with simultaneous requests
- ❖preferred when clear difference in rank between peripherals

Rotating priority (round-robin)

- ❖priority changed based on history of servicing
- ❖better distribution of servicing especially among peripherals with similar priority demands

Multilevel bus architectures

Don’t want one bus 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.

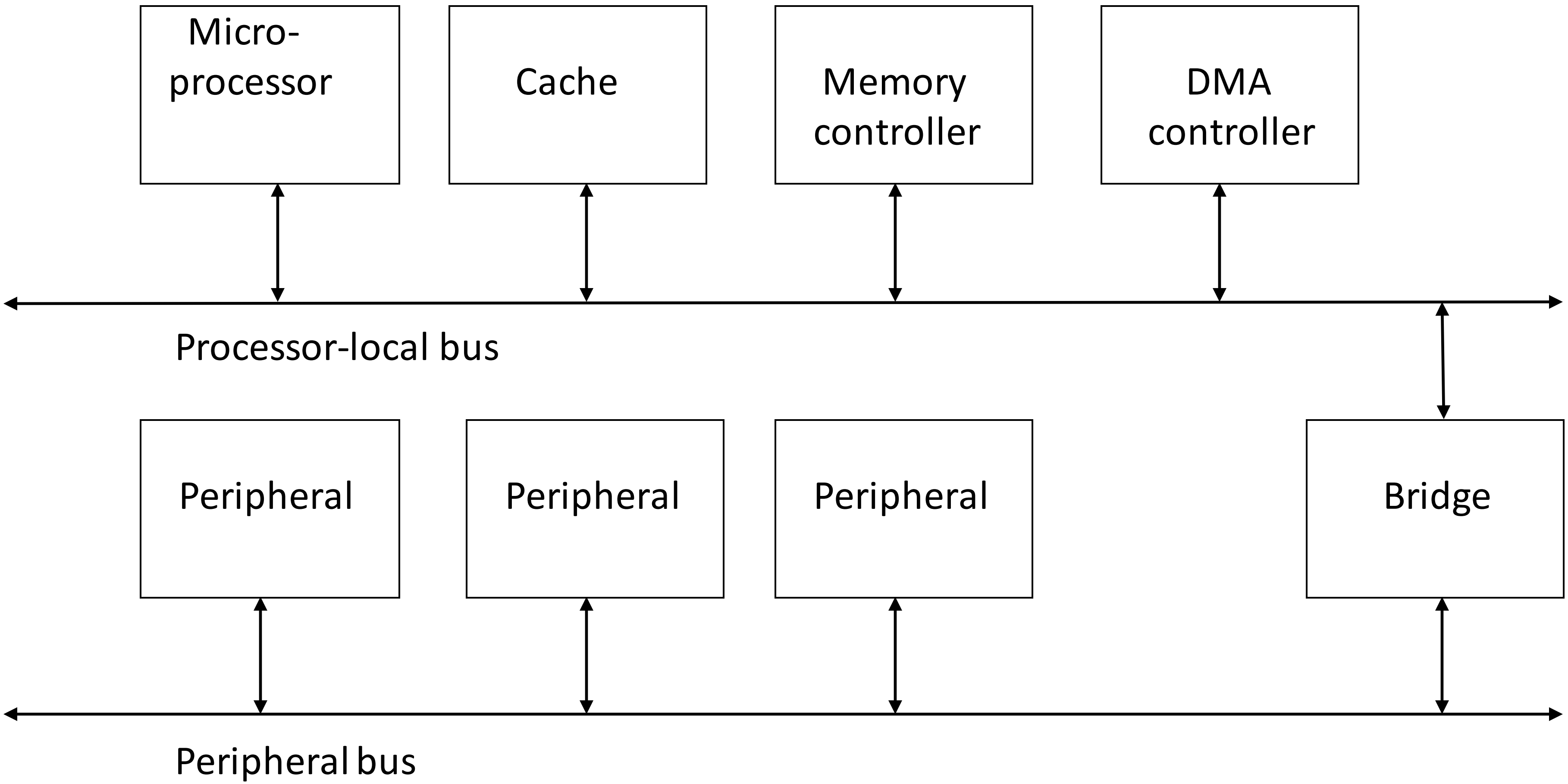
Peripheral bus

- 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



Advanced communication principles

Layering

- Break complexity of communication protocol into pieces easier to design and understand
- Lower levels provide services to higher level
  - Lower level might work with bits while higher level might work with packets of data
- Physical layer
  - Lowest level in hierarchy
  - Medium to carry data from one actor (device or node) to another

Parallel communication

- Physical layer capable of transporting multiple bits of data

Serial communication

- Physical layer transports one bit of data at a time

Wireless communication

- No physical connection needed for transport at physical layer

Parallel communication

- ❖ Multiple data, control, and possibly power wires
  - ❖ One bit per wire
- ❖ High data throughput with short distances
- ❖ Typically used when connecting devices on same IC or same circuit board
  - ❖ Bus must be kept short
    - ❖ long parallel wires result in high capacitance values which requires more time to charge/discharge
    - ❖ Data misalignment between wires increases as length increases
- ❖ Higher cost, bulky

Serial communication

Single data wire, possibly also control and power wires

Words transmitted one bit at a time

Higher data throughput with long distances

- ❖ Less average capacitance, so more bits per unit of time

Cheaper, less bulky

More complex interfacing logic and communication protocol

- ❖ Sender needs to decompose word into bits
- ❖ Receiver needs to recompose bits into word
- ❖ Control signals often sent on same wire as data increasing protocol complexity

Wireless communication

Infrared (IR)

- Electronic wave frequencies just below visible light spectrum
- Diode emits infrared light to generate signal
- Infrared transistor detects signal, conducts when exposed to infrared light
- Cheap to build
- Need line of sight, limited range

Radio frequency (RF)

- Electromagnetic wave frequencies in radio spectrum
- Analog circuitry and antenna needed on both sides of transmission
- Line of sight not needed, transmitter power determines range

Error detection and correction

Often part of bus protocol

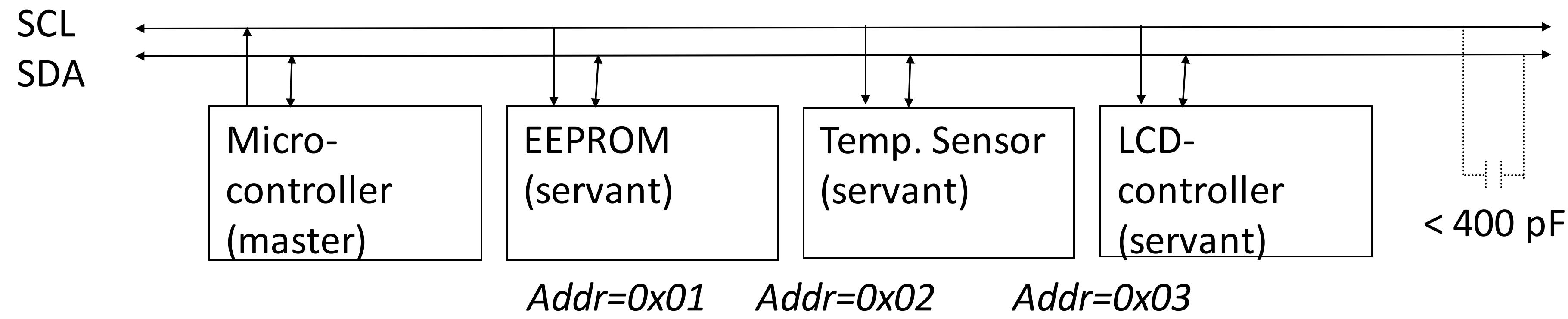
- Error detection: ability of receiver to detect errors during transmission
- Error correction: ability of receiver and transmitter to cooperate to correct problem
  - Typically done by acknowledgement/retransmission protocol
- Bit error: single bit is inverted
- Burst of bit error: consecutive bits received incorrectly
- Parity: extra bit sent with word used for error detection
  - Odd parity: data word plus parity bit contains odd number of 1's
  - Even parity: data word plus parity bit contains even number of 1's
  - Always detects single bit errors, but not all burst bit errors
- Checksum: extra word sent with data packet of multiple words
  - e.g., extra word contains XOR sum of all data words in packet

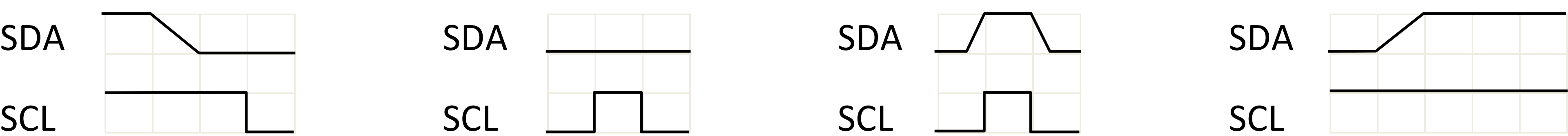
Serial protocols: I^2C

I^2C (Inter-IC)

- Two-wire serial bus protocol developed by Philips Semiconductors nearly 20 years ago.
- Enables peripheral ICs to communicate using simple communication hardware.
- Data transfer rates up to 100 kbits/s and 7-bit addressing possible in normal mode.
- 3.4 Mbits/s and 10-bit addressing in fast-mode.
- Common devices capable of interfacing to I^2C bus:
  - EPROMS, Flash, and some RAM memory, real-time clocks, watchdog timers, and microcontrollers

I2C bus structure



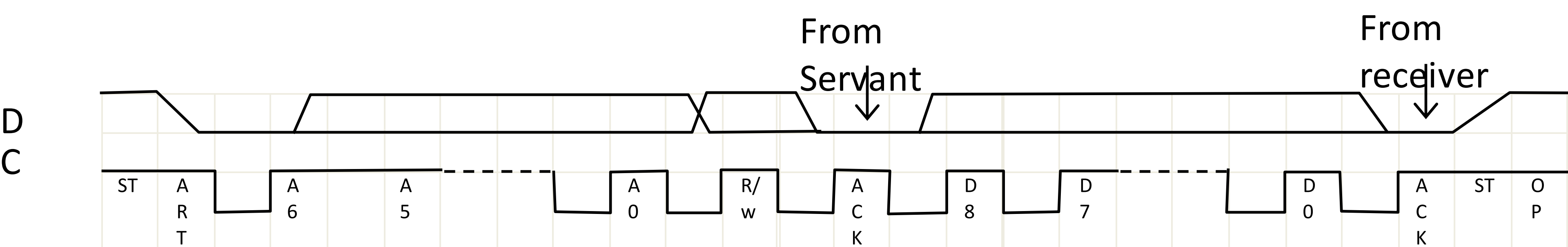


Start condition

Sending 0

Sending 1

Stop condition



Typical read/write cycle

Serial protocols: CAN

CAN (Controller area network)

- ❖ Protocol for real-time applications
- ❖ Developed by Robert Bosch GmbH
- ❖ Originally for communication among components of cars
- ❖ Applications now using CAN include:
  - ❖ elevator controllers, copiers, telescopes, production-line control systems, and medical instruments
- ❖ Data transfer rates up to 1 Mbit/s and 11-bit addressing
- ❖ Common devices interfacing with CAN:
  - ❖ 8051-compatible 8592 processor and standalone CAN controllers
- ❖ Actual physical design of CAN bus not specified in protocol
  - ❖ Requires devices to transmit/detect dominant and recessive signals to/from bus
  - ❖ e.g., ‘1’ = dominant, ‘0’ = recessive if single data wire used
  - ❖ Bus guarantees dominant signal prevails over recessive signal if asserted simultaneously

Serial protocols: FireWire

FireWire (a.k.a. I-Link, Lynx, IEEE 1394)

- ❖ High-performance serial bus developed by Apple Computer Inc.
- ❖ Designed for interfacing independent electronic components
  e.g., Desktop, scanner
- ❖ Data transfer rates from 12.5 to 400 Mbits/s, 64-bit addressing
- ❖ Plug-and-play capabilities
- ❖ Packet-based layered design structure
- ❖ Applications using FireWire include:
  disk drives, printers, scanners, cameras
- ❖ Capable of supporting a LAN similar to Ethernet
  64-bit address:
    - ❖10 bits for network ids, 1023 subnetworks
    - ❖6 bits for node ids, each subnetwork can have 63 nodes
    - ❖48 bits for memory address, each node can have 281 terabytes of distinct locations

Serial protocols: USB

USB (Universal Serial Bus)

- Easier connection between PC and monitors, printers, digital speakers, modems, scanners, digital cameras, joysticks, multimedia game equipment.

- 2 data rates:
  - 12 Mbps for increased bandwidth devices
  - 1.5 Mbps for lower-speed devices (joysticks, game pads)

- Tiered star topology can be used
  - One USB device (hub) connected to PC
    - hub can be embedded in devices like monitor, printer, or keyboard or can be standalone
  - Multiple USB devices can be connected to hub
  - Up to 127 devices can be connected like this

- USB host controller
  - Manages and controls bandwidth and driver software required by each peripheral
  - Dynamically allocates power downstream according to devices connected/disconnected

Parallel protocols: PCI Bus

PCI Bus (Peripheral Component Interconnect):

- ❖ High performance bus originated at Intel in the early 1990’s
- ❖ Standard adopted by industry and administered by PCISIG (PCI Special Interest Group)
- ❖ Interconnects chips, expansion boards, processor memory subsystems
- ❖ Data transfer rates of 127.2 to 508.6 Mbits/s and 32-bit addressing
  - ❖ Later extended to 64-bit while maintaining compatibility with 32-bit schemes
- ❖ Synchronous bus architecture
- ❖ Multiplexed data/address lines

Parallel protocols: ARM Bus

ARM Bus

- ❖ Designed and used internally by ARM Corporation
- ❖ Interfaces with ARM line of processors
- ❖ Many IC design companies have own bus protocol
- ❖ Data transfer rate is a function of clock speed
  - ❖ If clock speed of bus is X, transfer rate = 16 x X bits/s
- ❖ 32-bit addressing

Wireless protocols: IrDA

IrDA

- ❖ Protocol suite that supports short-range point-to-point infrared data transmission
- ❖ Created and promoted by the Infrared Data Association (IrDA)
- ❖ Data transfer rate of 9.6 kbps and 4 Mbps
- ❖ IrDA hardware deployed in notebook computers, printers, PDAs, digital cameras, public phones, cell phones
- ❖ Lack of suitable drivers has slowed use by applications
- ❖ Windows 2000/98 now include support
- ❖ Becoming available on popular embedded OS’s

Wireless protocols: Bluetooth

Bluetooth

- ❖ New, global standard for wireless connectivity
- ❖ Based on low-cost, short-range radio link
- ❖ Connection established when within 10 meters of each other
- ❖ No line-of-sight required
  - ❖ e.g., Connect to printer in another room

Wireless Protocols: IEEE 802.11

IEEE 802.11

- Proposed standard for wireless LANs

- Specifies parameters for PHY and MAC layers of network

  PHY layer

  - physical layer
  - handles transmission of data between nodes
  - provisions for data transfer rates of 1 or 2 Mbps
  - operates in 2.4 to 2.4835 GHz frequency band (RF)
  - or 300 to 428,000 GHz (IR)

  MAC layer

  - medium access control layer
  - protocol responsible for maintaining order in shared medium
  - collision avoidance/detection