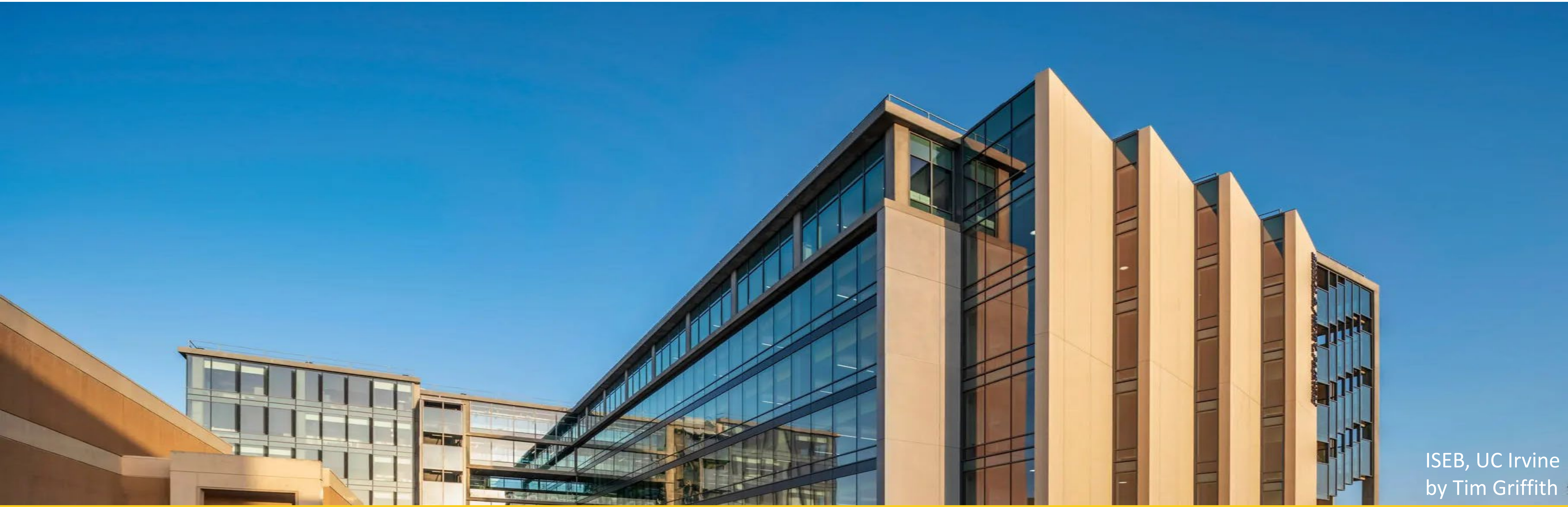
Lecture 5:

Embedded Systems

Sitao Huang, sitaoh@uci.edu

October 23, 2023

Sources: Prof. Deming Chen, UIUC ECE 527 “System-on-Chip Design”;
Vahid/Givargis, “Embedded System Design: A Unified Hardware/Software Introduction”



ISEB, UC Irvine
by Tim Griffith

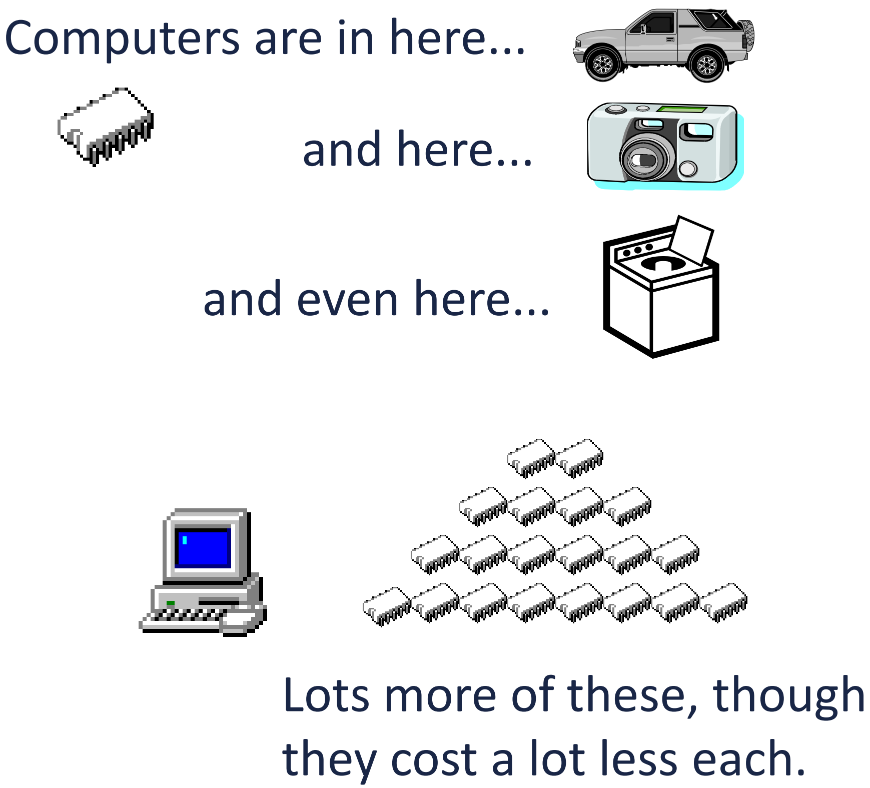
Outline

- Embedded System Overview
- Processor and Memory Overview
- Peripherals
- Interfacing
- Embedded Software (ESW) Design
- Major ESW Issues

Embedded System Overview

Embedded Computing Systems

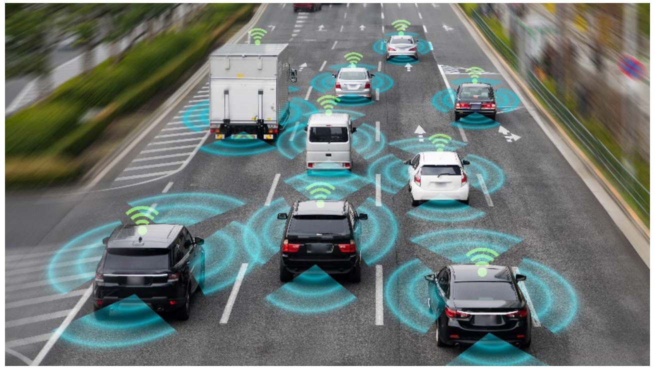
- Computing systems embedded within electronic devices
- Hard to define. Nearly any computing system other than a desktop computer
- Billions of units produced yearly, versus millions of desktop units
- Perhaps 50 per household and per automobile

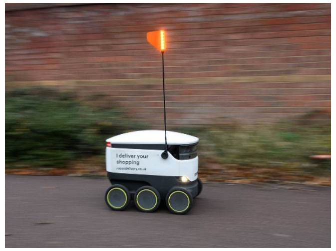


A “Short List” of Embedded Systems

Anti-lock brakes
Auto-focus cameras
Automatic teller machines
Automatic toll systems
Automatic transmission
Avionic systems
Battery chargers
Camcorders
Cell phones
Cell-phone base stations
Cordless phones
Cruise control
Curbside check-in systems
Digital cameras
Disk drives
Electronic card readers
Electronic instruments
Electronic toys/games
Factory control
Fax machines
Fingerprint identifiers
Home security systems
Life-support systems
Medical testing systems

Modems
MPEG decoders
Network cards
Network switches/routers
On-board navigation
Pagers
Photocopiers
Point-of-sale systems
Portable video games
Printers
Satellite phones
Scanners
Smart ovens/dishwashers
Speech recognizers
Stereo systems
Teleconferencing systems
Televisions
Temperature controllers
Theft tracking systems
TV set-top boxes
VCR’s, DVD players
Video game consoles
Video phones
Washers and dryers





And the list goes on and on

Some Common Characteristics of Embedded Systems

- Single-Functioned
  - Executes a single program, repeatedly
- Tightly-constrained
  - Low cost, low power, small, fast, etc.
- Reactive and real-time
  - Continually reacts to changes in the system’s environment
  - Must compute certain results in real-time without delay

Embedded System Functionality

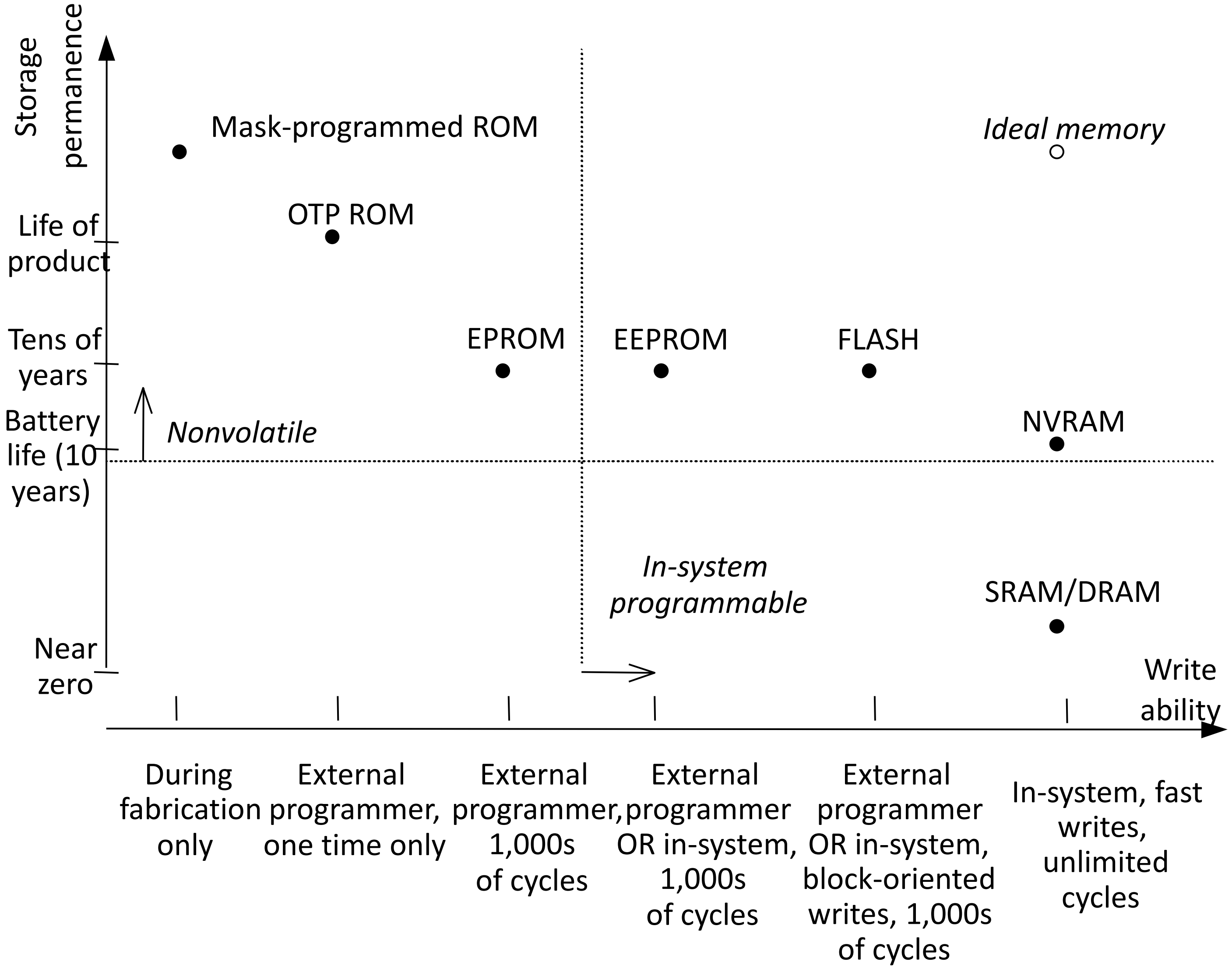
- Processing
  - Transformation of data
  - Implemented using processors
- Storage
  - Retention of data
  - Implemented using memory
- Peripheral
  - Connecting to the real world
  - Timers, UART, ADC, DAC, etc.
- Communication or interfacing
  - Transfer of data between processors, memories and peripherals
  - Implemented using buses and others

Embedded System Functionality

- Processing
  - Transformation of data
  - Implemented using processors
- **Storage**
  - **Retention of data**
  - **Implemented using memory**
- Peripheral
  - Connecting to the real world
  - Timers, UART, ADC, DAC, etc.
- Communication or interfacing
  - Transfer of data between processors, memories and peripherals
  - Implemented using buses and others

Memory: Write ability / Storage Permanence

- Traditional ROM/RAM distinctions
  - ROM
    - read only, bits stored without power
  - RAM
    - read and write, lose stored bits without power
- Traditional distinctions blurred
  - Advanced ROMs can be written to
    - e.g., EEPROM
  - Advanced RAMs can hold bits without power
    - e.g., NVRAM
- Write ability
  - Manner and speed a memory can be written
- Storage permanence
  - ability of memory to hold stored bits after they are written



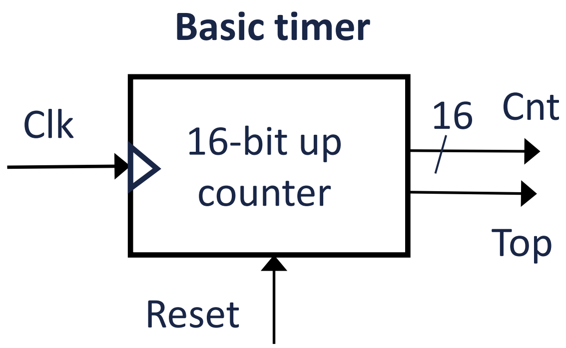
Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).

Embedded System Functionality

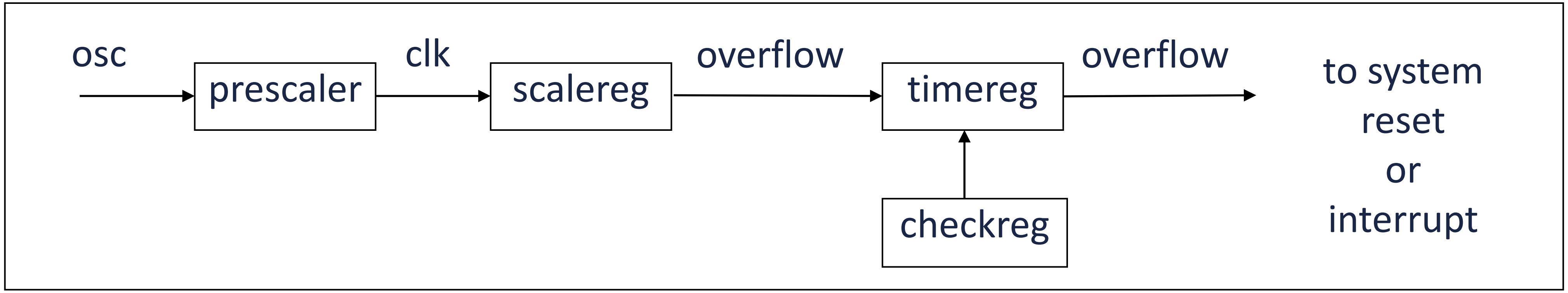
- Processing
  - Transformation of data
  - Implemented using processors
- Storage
  - Retention of data
  - Implemented using memory
- **Peripheral**
  - **Connecting to the real world**
  - **Timers, UART, ADC, DAC, etc.**
- Communication or interfacing
  - Transfer of data between processors, memories and peripherals
  - Implemented using buses and others

Peripheral: Timers, Counters, Watch Dog Timers

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting clock pulses
  - E.g., let Clk period be 10 ns
  - And we count 20,000 Clk pulses
  - Then 200 microseconds have passed
  - 16-bit counter would count up to 65,535*10 ns = 655.35 microsec., resolution = 10 ns
  - Top: indicates top count reached, wrap-around



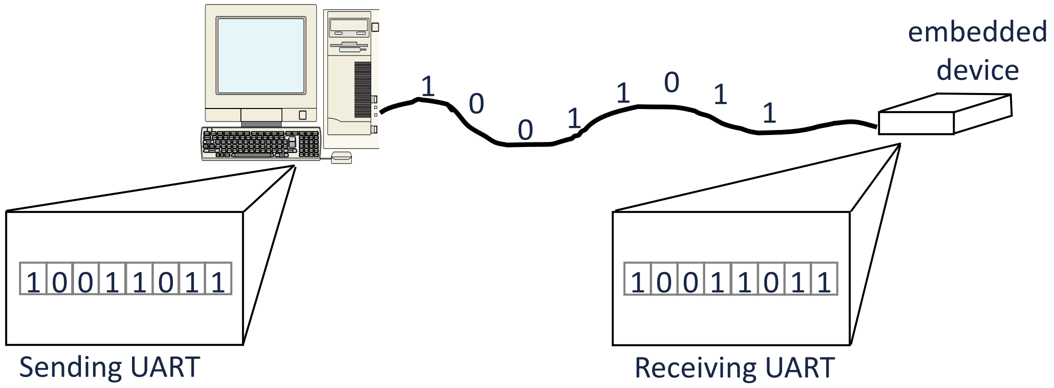
Peripheral: Timers, Counters, Watch Dog Timers

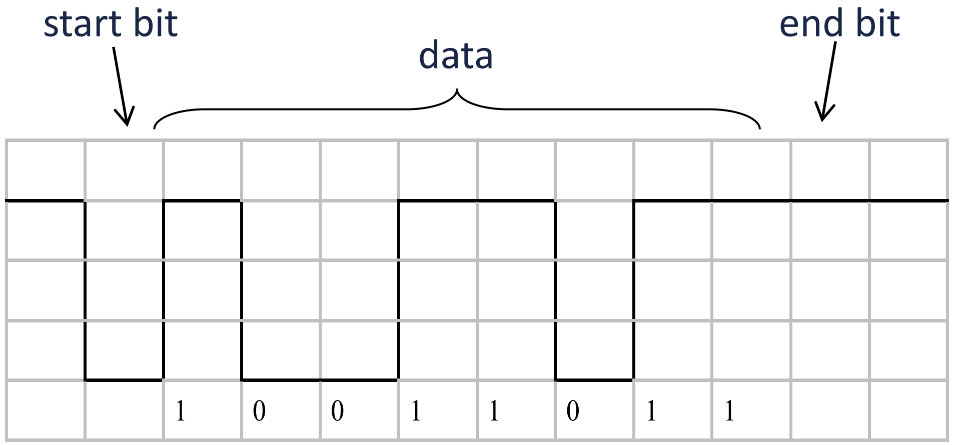


- Must reset timer every X time unit, otherwise timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - *timereg* value = $2*(2^{16}-1) - X = 131070 - X$
  - For 2 min., X = 120,000 microsec.

Peripheral: Serial Transmission Using UART

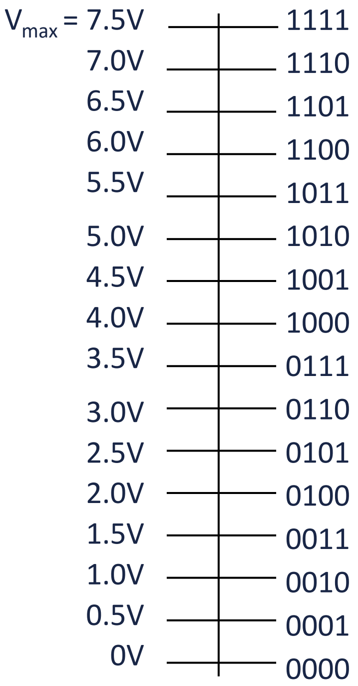
- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher



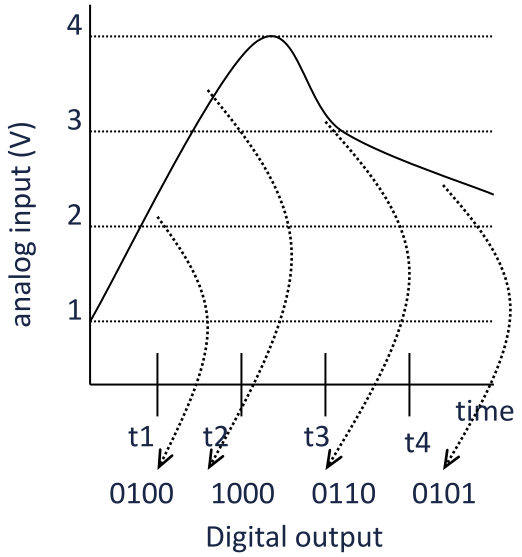


Peripheral: DAC and ADC

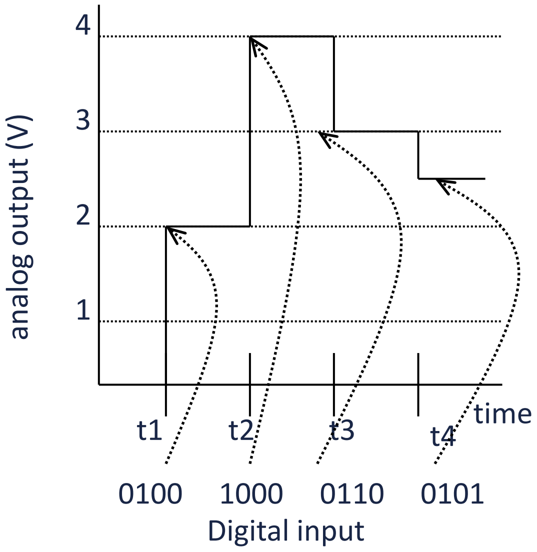
- Analog-to-Digital Converters (ADC)



proportionality



analog to digital



digital to analog

Peripheral: DAC and ADC

$$\frac{e}{V_{\max}} = \frac{d}{2^n - 1}$$

- DAC is simpler, digital arithmetic
- ADC is more difficult
  - No simple analog circuit to compute *d* from *e*
  - One solution:
    - It contains a DAC
    - ADC guesses an encoding *d* and evaluates its guess by inputting *d* into the DAC
    - Compare the generated analog output *e'* from DAC with *e* using an analog comparator
    - Use a *binary-search* method and do this until a match is found

Peripheral: DAC and ADC

- Analog-to-Digital conversion using successive approximation

Given an analog input signal whose voltage should range from 0 to 15 volts, and an 8-bit digital encoding, calculate the correct encoding for 5 volts. Then trace the successive-approximation approach to find the correct encoding.

$5/15 = d/(2^8-1)$
$d = 85$

Encoding: 01010101

Successive-approximation method

$\frac{1}{2}(V_{max} - V_{min}) = 7.5$ volts
$V_{max} = 7.5$ volts.

0	0	0	0	0	0	0	0

$\frac{1}{2}(7.5 + 0) = 3.75$ volts
$V_{min} = 3.75$ volts.

0	1	0	0	0	0	0	0

$\frac{1}{2}(7.5 + 3.75) = 5.63$ volts
$V_{max} = 5.63$ volts

0	1	0	0	0	0	0	0

$\frac{1}{2}(5.63 + 3.75) = 4.69$ volts
$V_{min} = 4.69$ volts.

0	1	0	1	0	0	0	0

$\frac{1}{2}(5.63 + 4.69) = 5.16$ volts
$V_{max} = 5.16$ volts.

0	1	0	1	0	0	0	0

$\frac{1}{2}(5.16 + 4.69) = 4.93$ volts
$V_{min} = 4.93$ volts.

0	1	0	1	0	1	0	0

$\frac{1}{2}(5.16 + 4.93) = 5.05$ volts
$V_{max} = 5.05$ volts.

0	1	0	1	0	1	0	0

$\frac{1}{2}(5.05 + 4.93) = 4.99$ volts

0	1	0	1	0	1	0	1

Embedded System Functionality

- Processing
  - Transformation of data
  - Implemented using processors
- Storage
  - Retention of data
  - Implemented using memory
- Peripheral
  - Connecting to the real world
  - Timers, UART, ADC, DAC, etc.
- **Communication or interfacing**
  - **Transfer of data between processors, memories and peripherals**
  - **Implemented using buses and others**

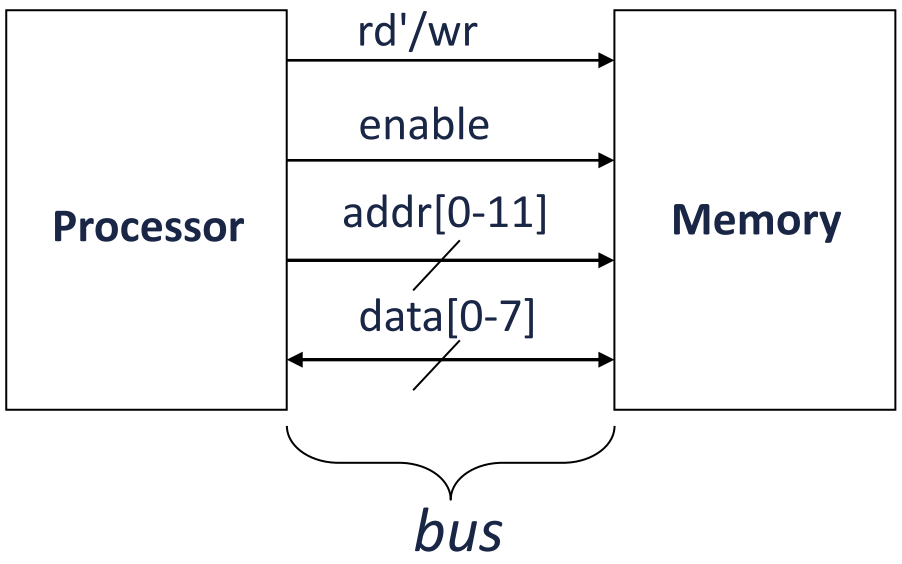
Interfacing: A Simple Bus

Wires

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

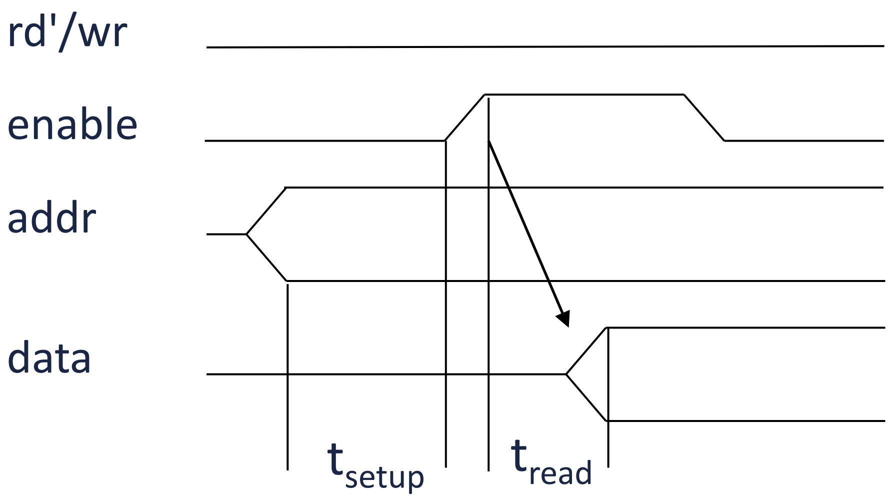
Bus

- 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

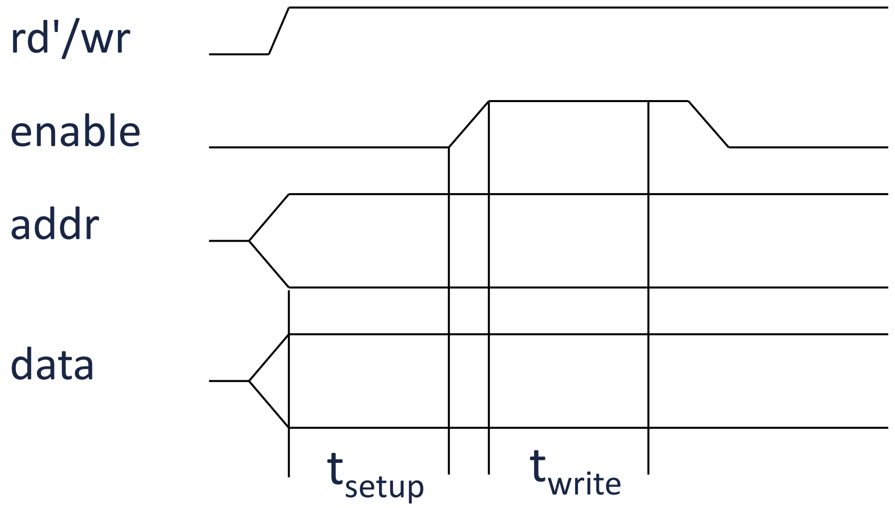


Interfacing: Timing Diagrams

- 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

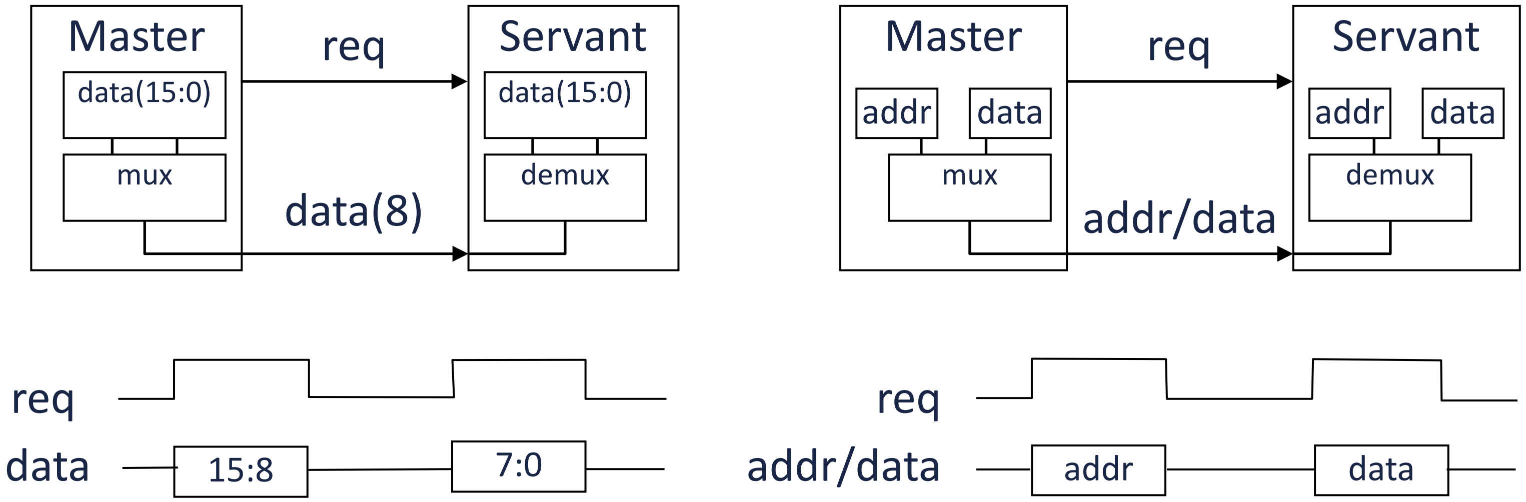


write protocol

Interfacing: Basic Protocol Concepts

- Actor: master initiates, servant (slave) responds
- 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

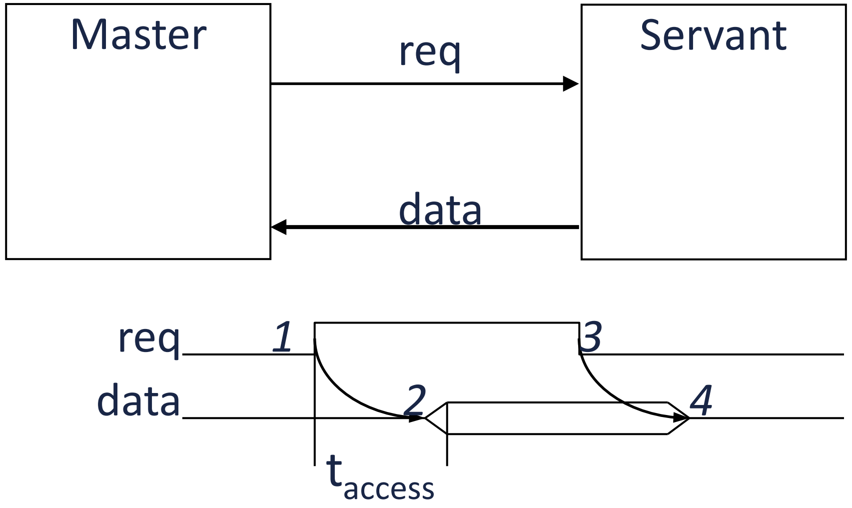
Time-multiplexed data transfer



data serializing

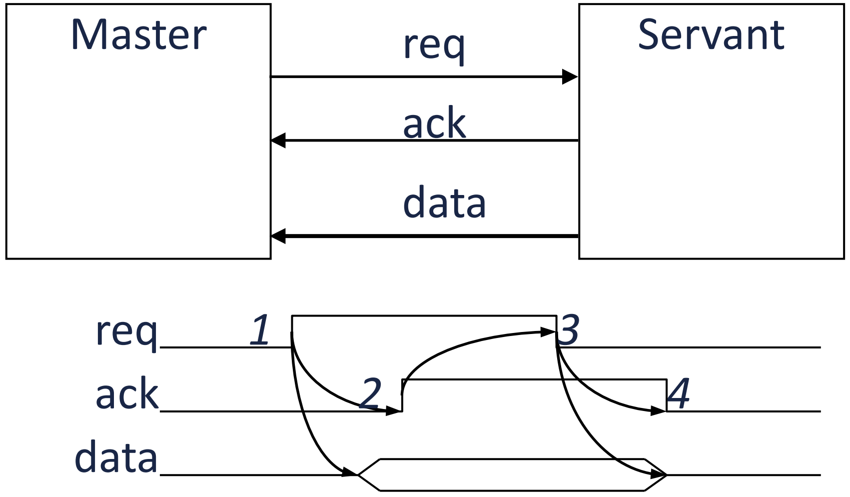
address/data muxing

Interfacing: Basic Protocol Concepts



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

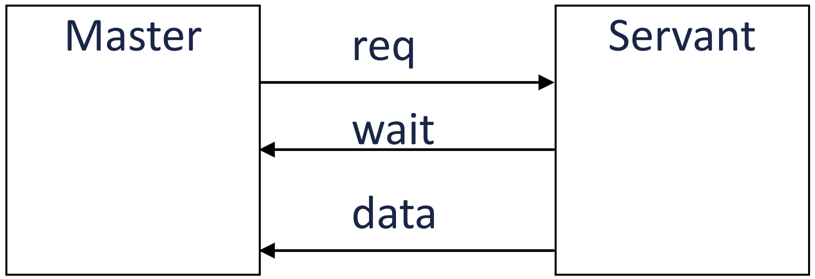


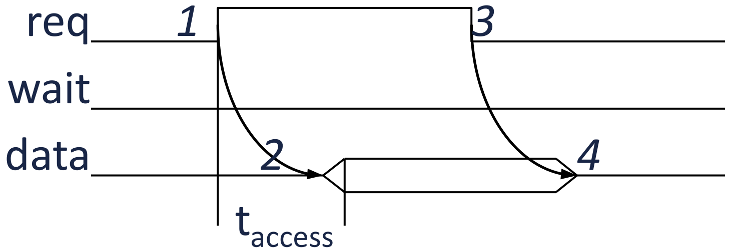
1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

Handshake protocol

Interfacing: Basic Protocol Concepts

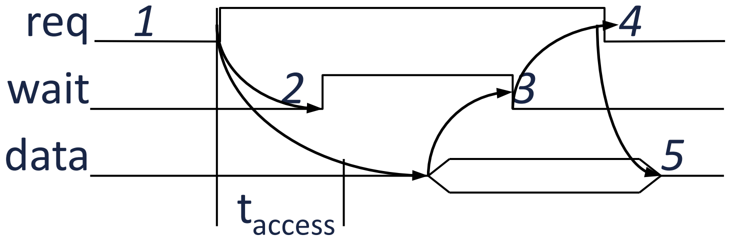
- A strobe/handshake compromise





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

Fast-response case



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

Slow-response case

Microprocessor Interfacing: I/O Addressing

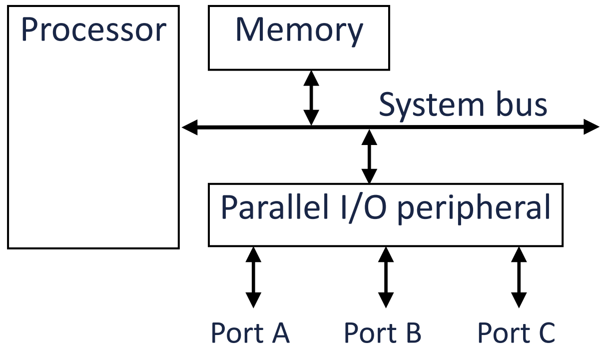
A microprocessor communicates with other devices using some of its pins

- 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;` -- 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

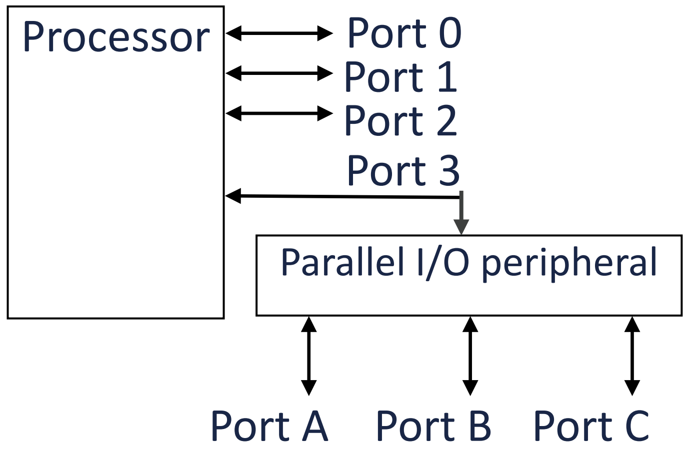
Microprocessor Interfacing: Compromises/Extensions

- 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

Interfacing: Types of Bus-based I/O

Memory-Mapped I/O and Standard I/O

Processor talks to both memory and peripherals using same bus – two ways to talk to peripherals

- Memory-mapped I/O
  - Peripheral registers occupy addresses in 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

Interfacing: 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 much smaller than address space then high-order address bits can be ignored
    - smaller and/or faster comparators

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.

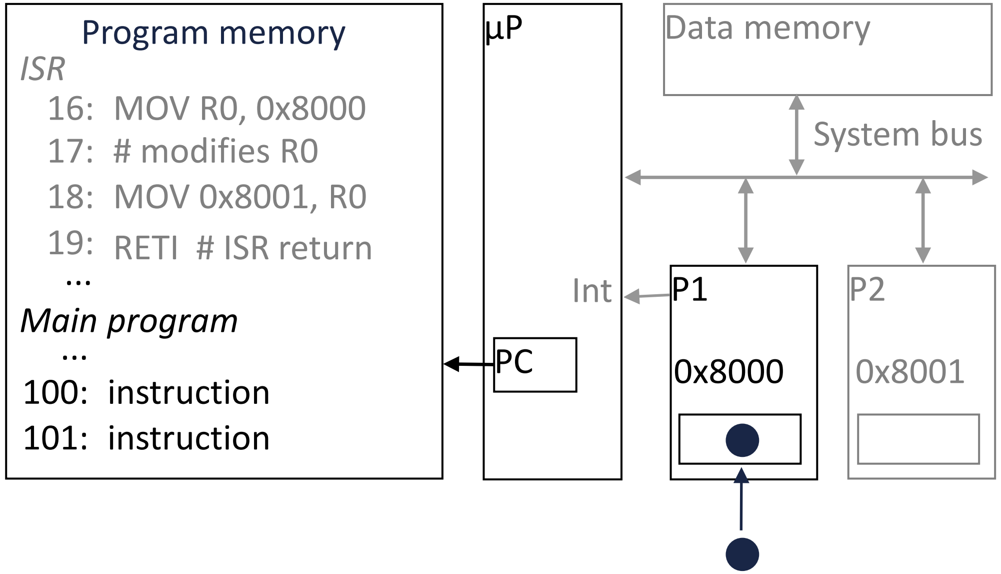
4(b): After being read, P1 de-asserts *Int*.

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

Interrupt-Driven I/O Using Fixed ISR Location

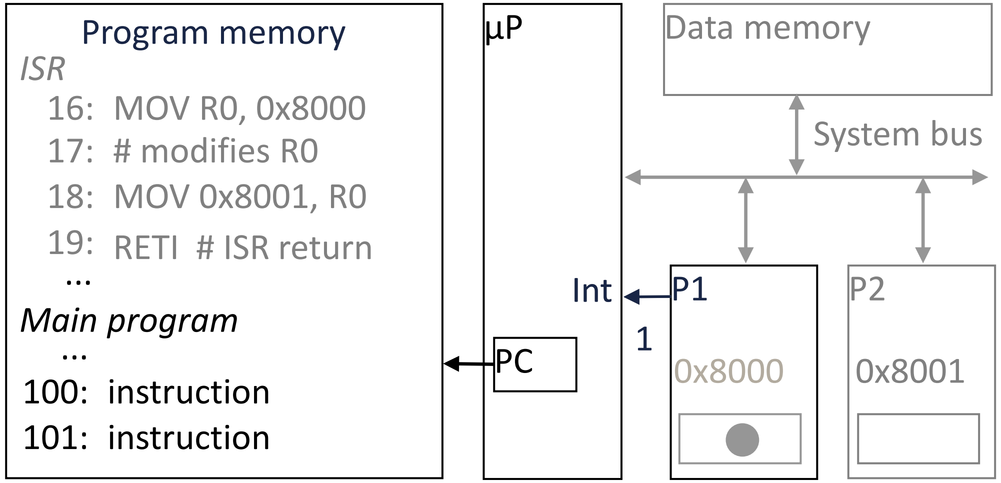
1(a): μP is executing its main program

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



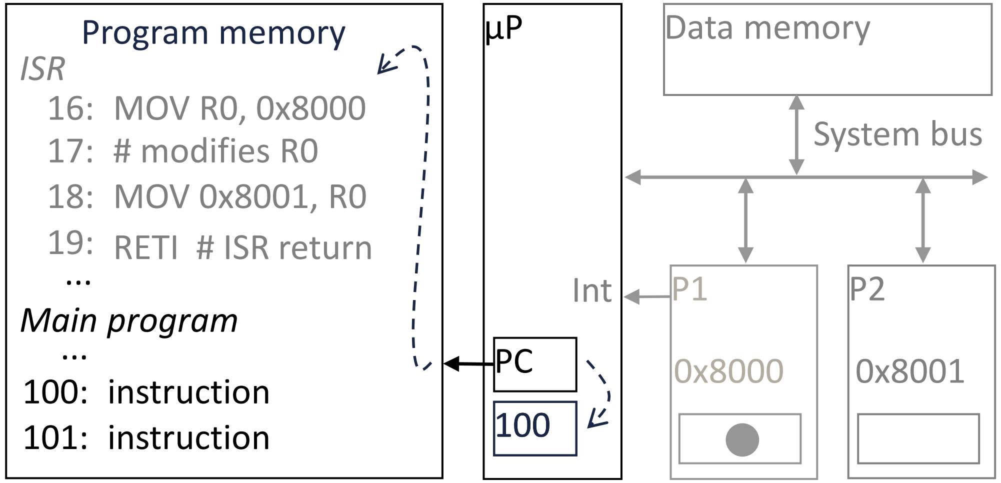
Interrupt-Driven I/O Using Fixed ISR Location

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



Interrupt-Driven I/O Using Fixed ISR Location

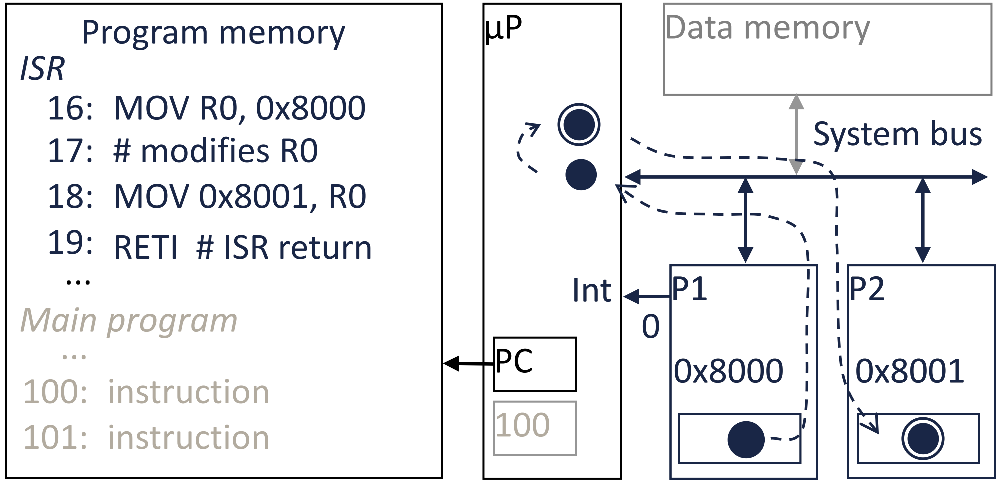
3: After completing instruction at 100, μP sees *Int* asserted, saves the PC's value of 100, and sets PC to the ISR fixed location of 16.



Interrupt-Driven I/O Using Fixed ISR Location

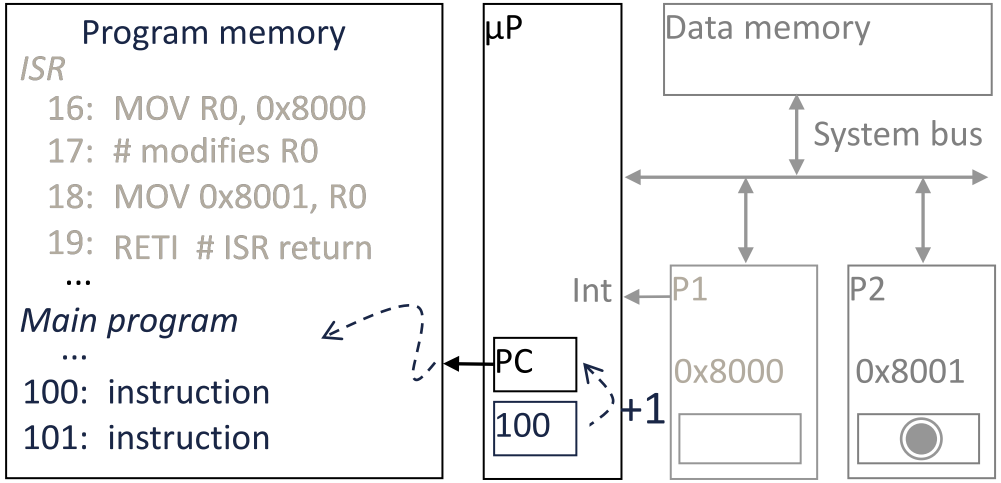
4(a): The ISR reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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.

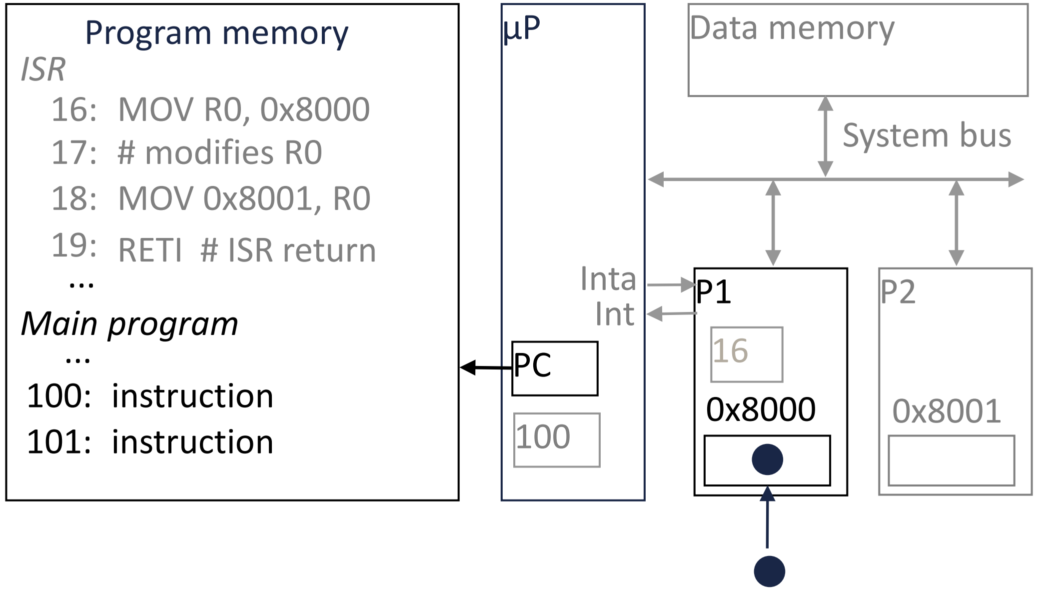
5(a): **µP jumps to the address on the bus (16)**. The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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

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

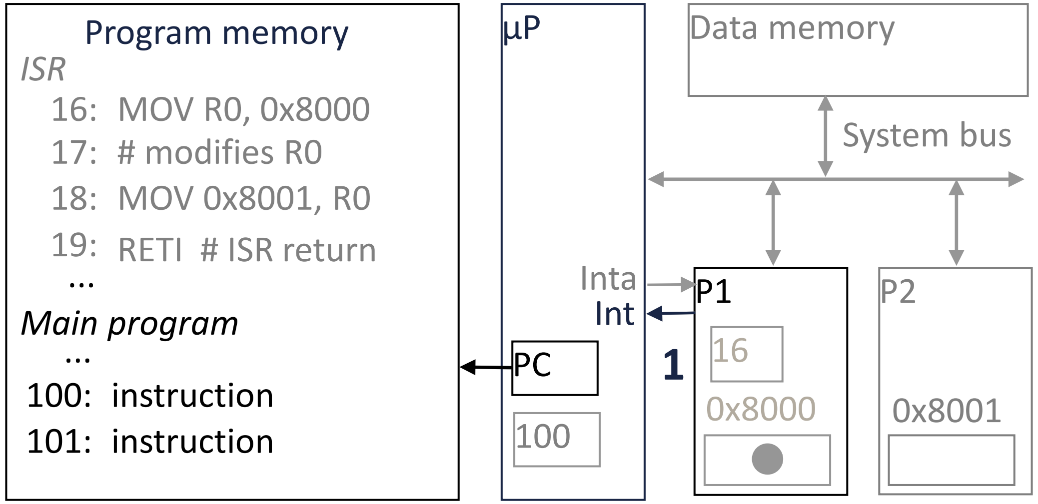
Interrupt-Driven I/O Using Vectored Interrupt

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



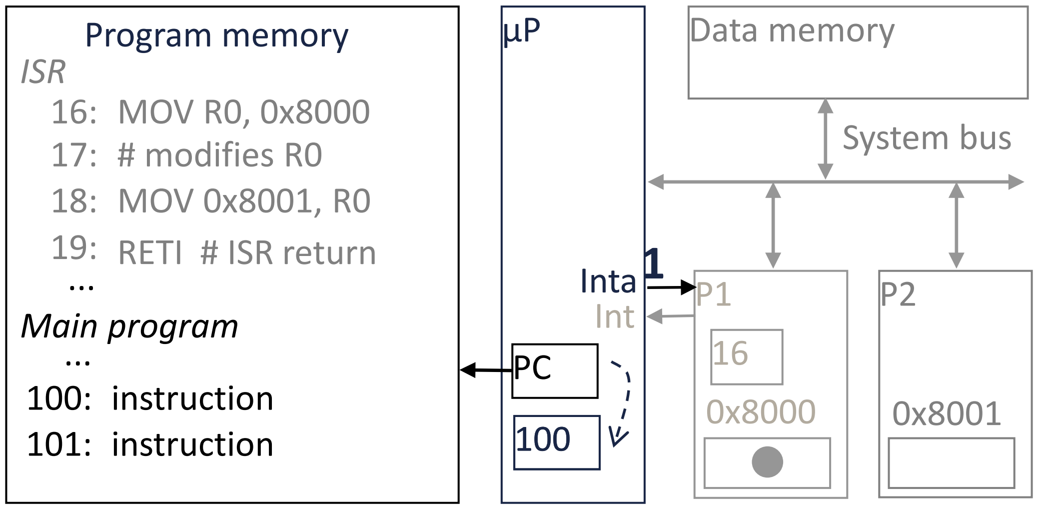
Interrupt-Driven I/O Using Vectored Interrupt

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



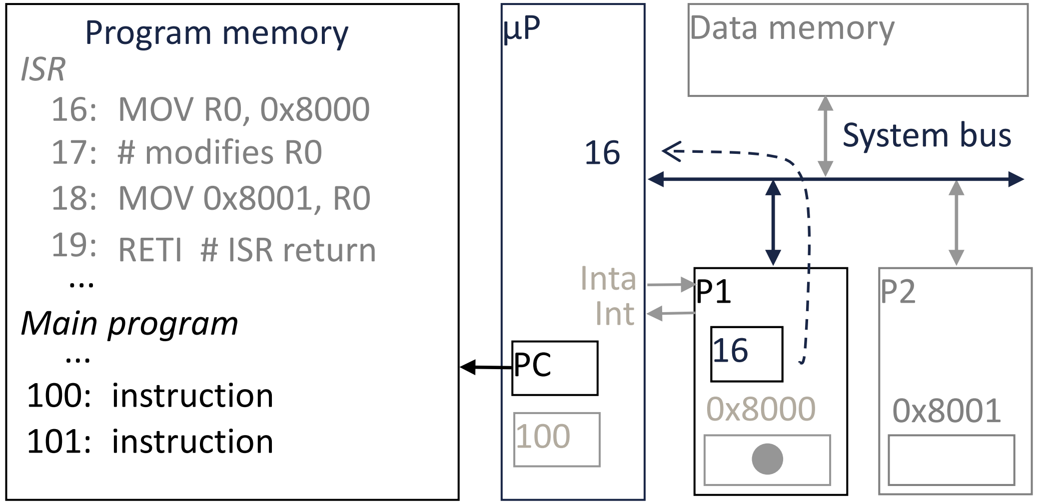
Interrupt-Driven I/O Using Vectored Interrupt

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

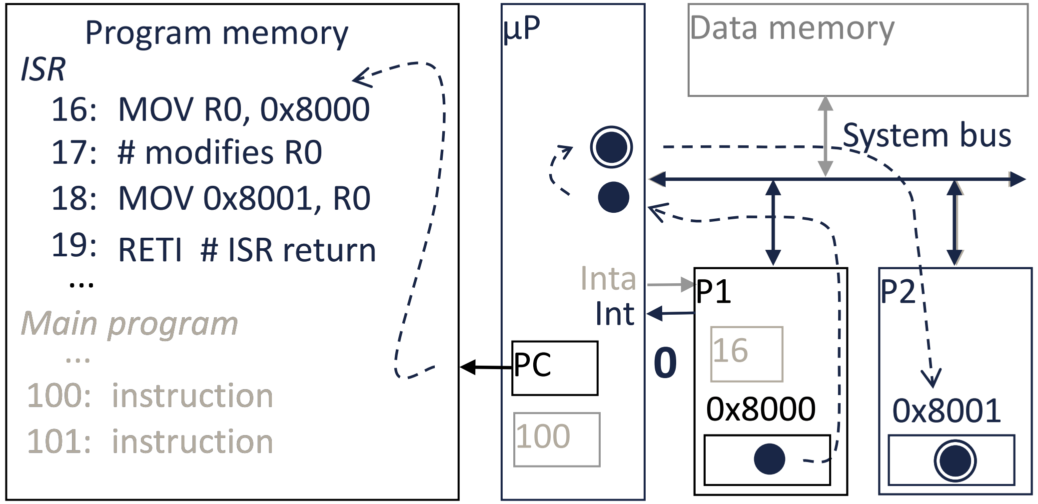
4: P1 detects *Inta* and puts **interrupt address vector 16** on the data bus



Interrupt-Driven I/O Using Vectored Interrupt

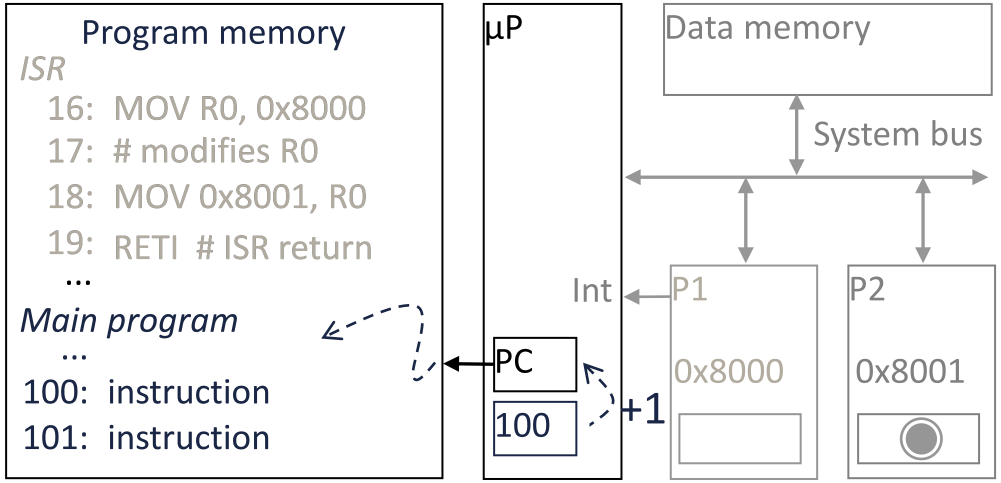
5(a): PC jumps to the address on the bus (16). The ISR there reads data from 0x8000, modifies the data, and writes the resulting data to 0x8001.

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.

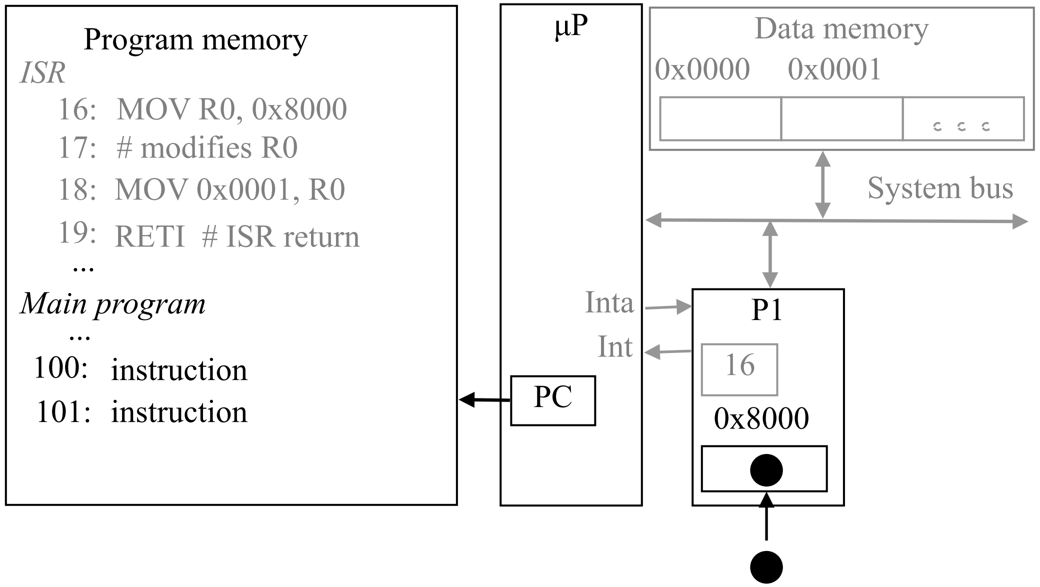
5(b): After being read, P1 deasserts *Int*.

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

Peripheral to Memory Transfer *without* DMA, Using Vectored Interrupt

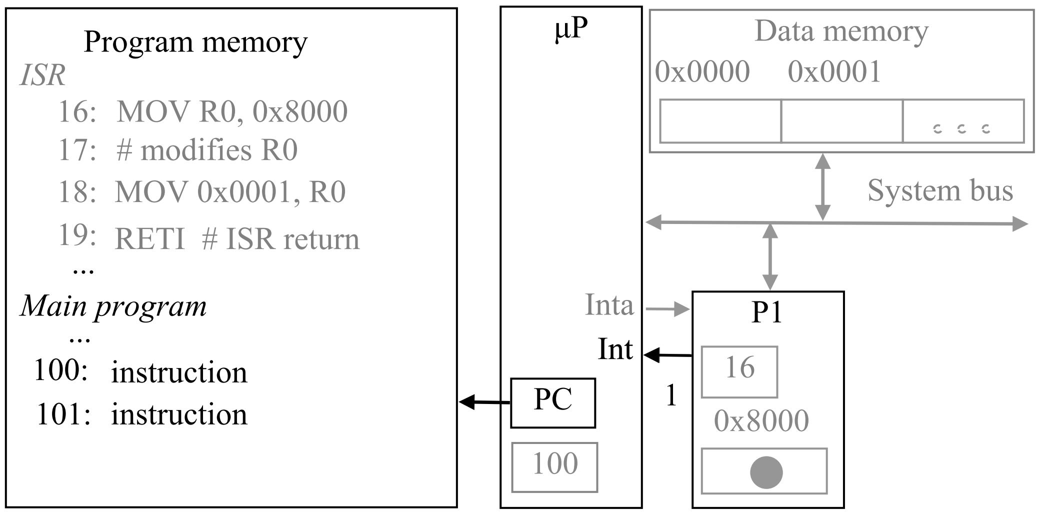
1(a): μP is executing its main program

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



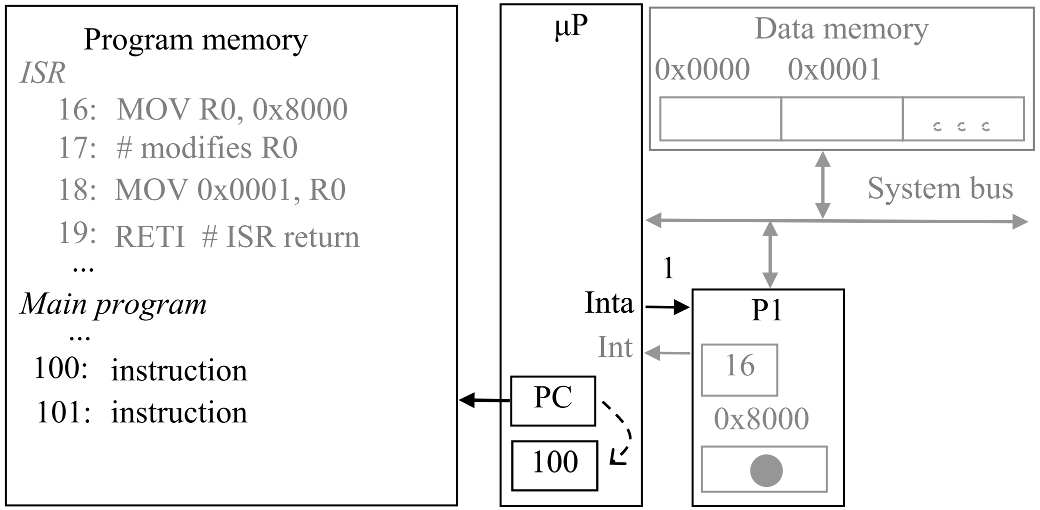
Peripheral to Memory Transfer *without* DMA, Using Vectored Interrupt

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



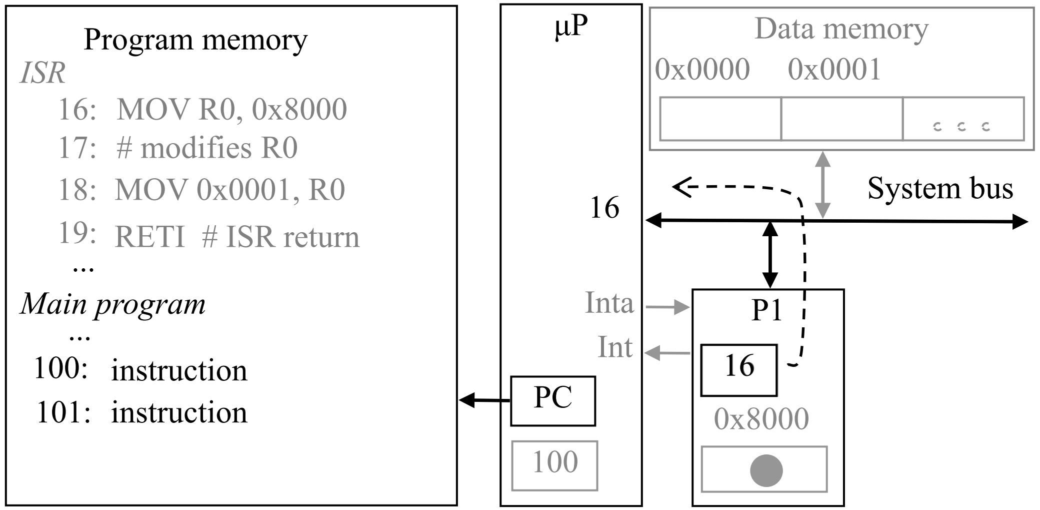
Peripheral to Memory Transfer *without* DMA, Using Vectored Interrupt

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

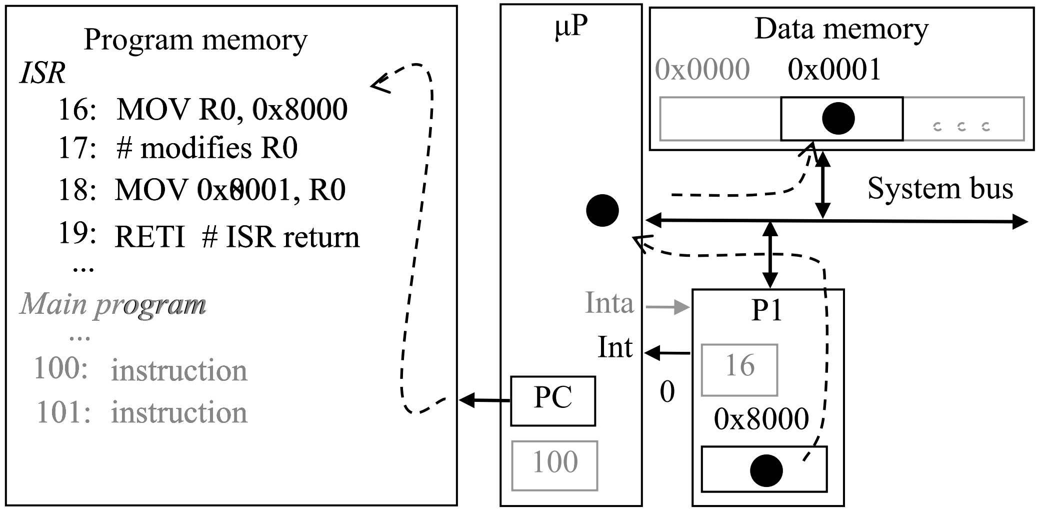
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



Peripheral to Memory Transfer *without* DMA, Using Vectored Interrupt

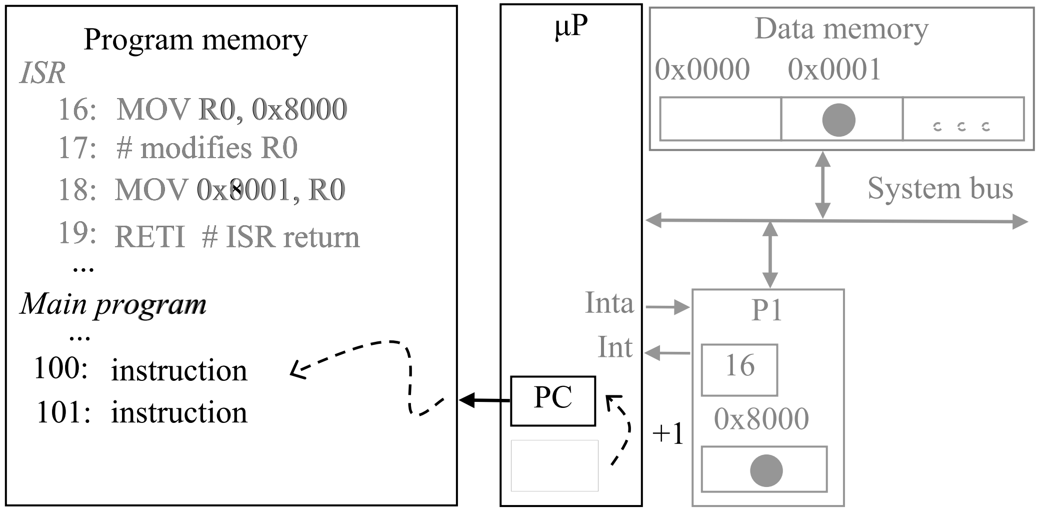
5(a): μP jumps to the address on the bus (16). The ISR there reads data from 0x8000 and then writes it to 0x0001, which is in memory.

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



Peripheral to Memory Transfer *without* DMA, Using Vectored Interrupt

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 ctrl* 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.

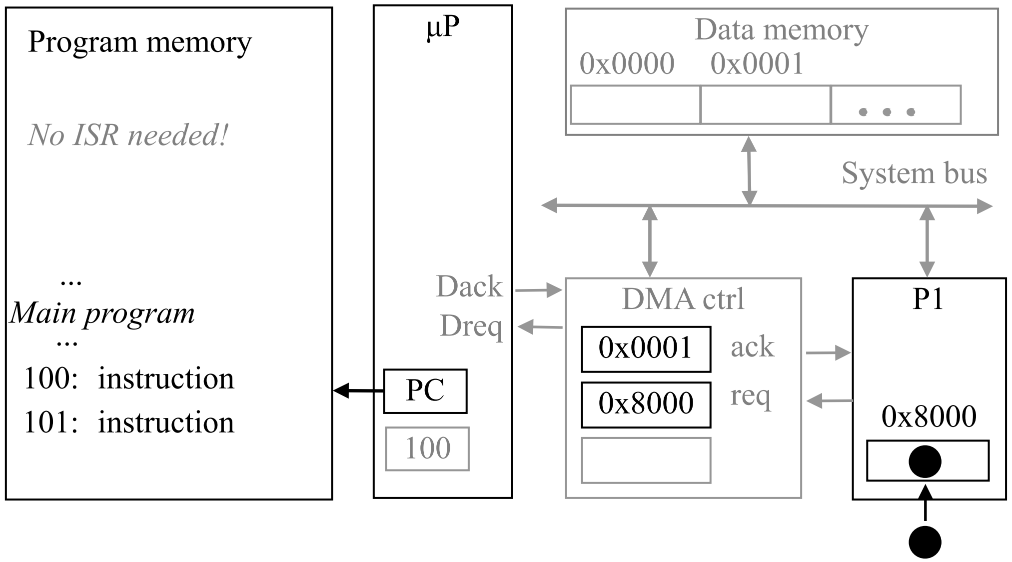
7(a): µP de-asserts *Dack* and resumes control of the bus.

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

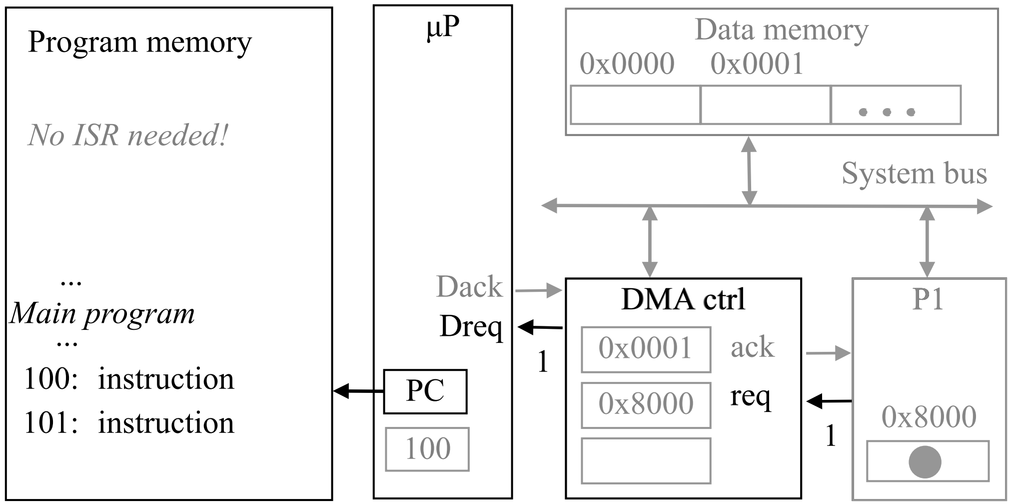
1(b): P1 receives input data in a register with address 0x8000.



Peripheral to Memory Transfer *with* DMA

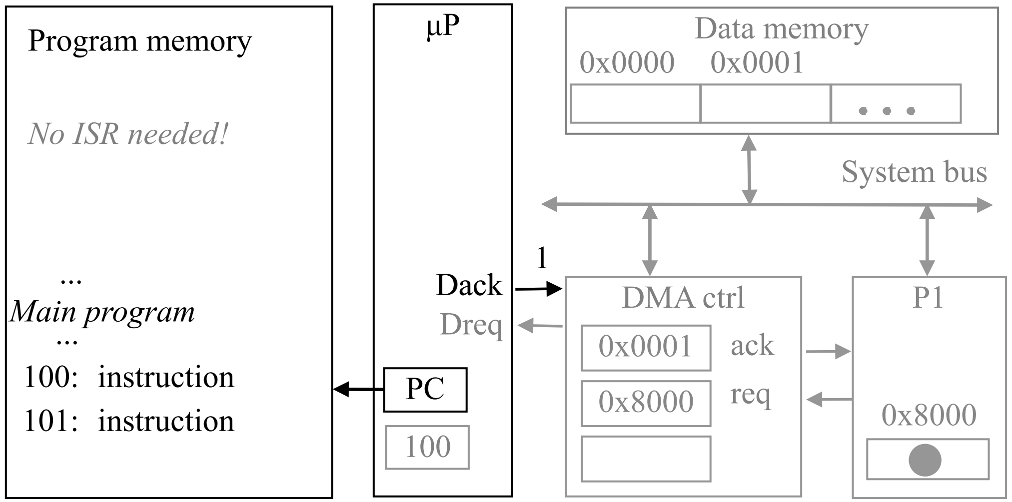
2: P1 asserts *req* to request servicing
by DMA ctrl.

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



Peripheral to Memory Transfer *with* DMA

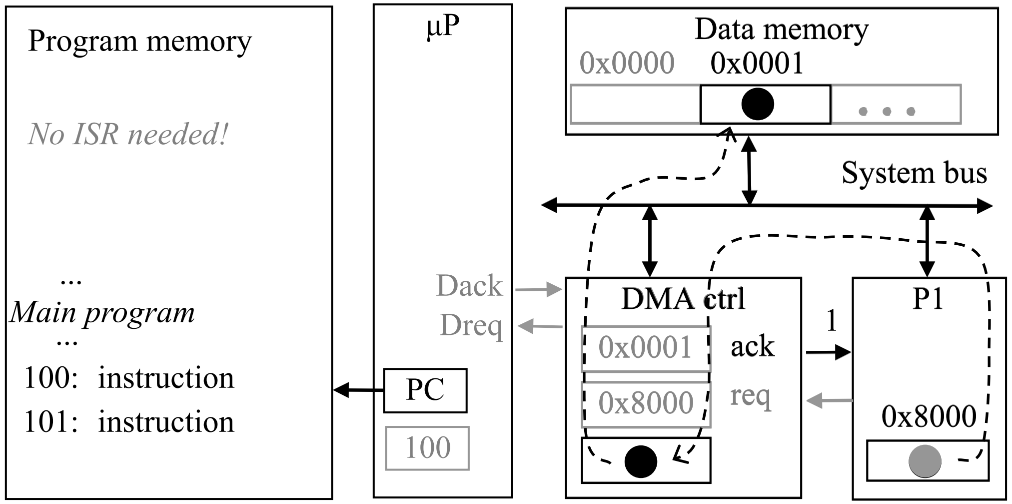
4: After executing instruction 100, μP sees *Dreq* asserted, releases the system bus, asserts *Dack*, and resumes execution, μP stalls only if it needs the system bus to continue executing.



Peripheral to Memory Transfer *with* DMA

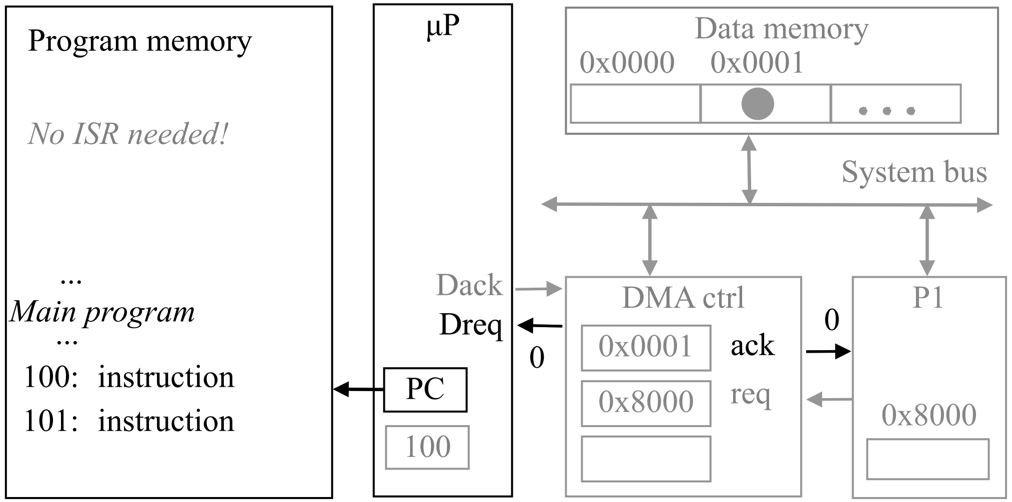
5: DMA ctrl (a) asserts ack, (b) reads data from 0x8000, and (c) writes that data to 0x0001.

(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.

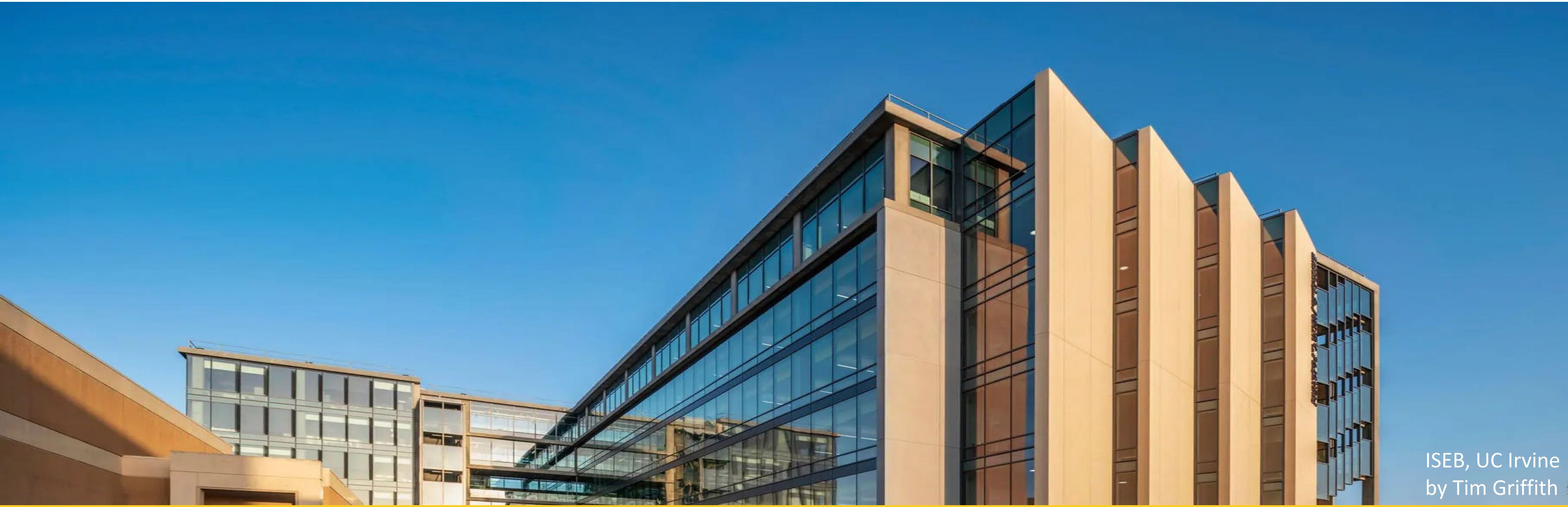


Reference

- Frank Vahid, Tony D. Givargis, “Embedded System Design: A Unified Hardware / Software Introduction”, 2001.

EECS 298:
System-on-Chip Design

Sitao Huang, sitaoh@uci.edu


ISEB, UC Irvine
by Tim Griffith