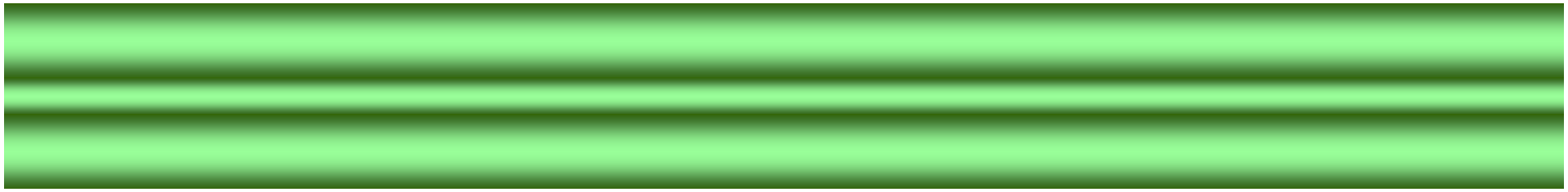
EE414 Embedded Systems

Ch 6. Interfacing

Part 4/4: DMA & Arbitration



Byung Kook Kim
School of Electrical Engineering
Korea Advanced Institute of Science and Technology

Overview

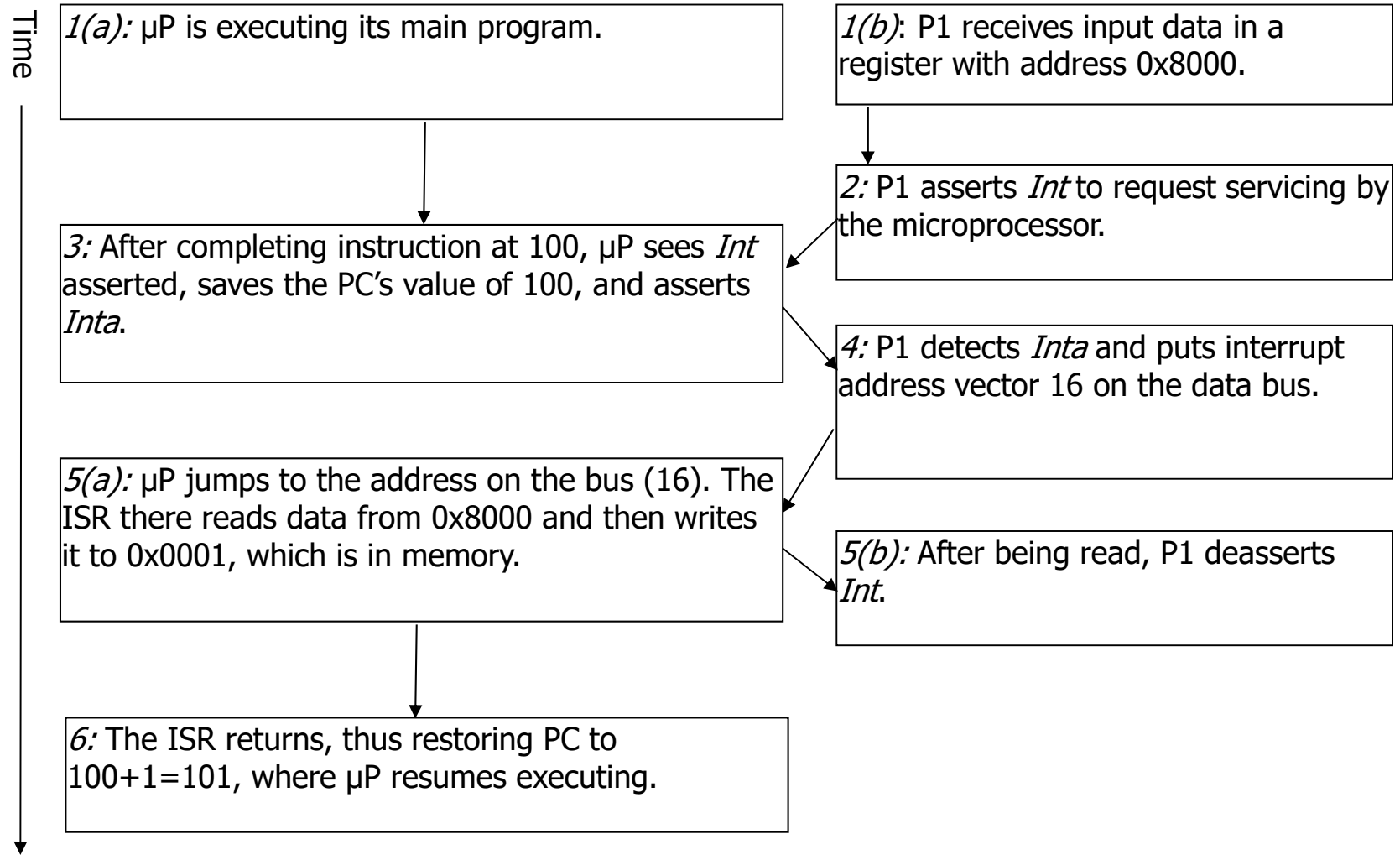
Part D. DMA and Arbitration

- 6.9 Direct Memory Access (DMA)
- 6.10 Arbitration
- 6.11 Multilevel Bus Architectures

6.9 Direct Memory Access

- Handling large I/O data with Buffering (e.g. disk read)
  - Data accumulated in peripherals are commonly buffered.
  - Buffering: Temporarily storing data in memory before processing
- Interrupt
  - Microprocessor could handle this with *ISR*
    - Storing and restoring microprocessor state inefficient
    - Regular program must wait
- **DMA controller** *is more efficient*
  - Separate single-purpose processor
  - Microprocessor releases control of system bus to DMA controller
  - Microprocessor can meanwhile execute its regular program.
  - Adv:
    - No inefficient storing and restoring state due to ISR call
    - Regular program need not wait unless it requires the system bus

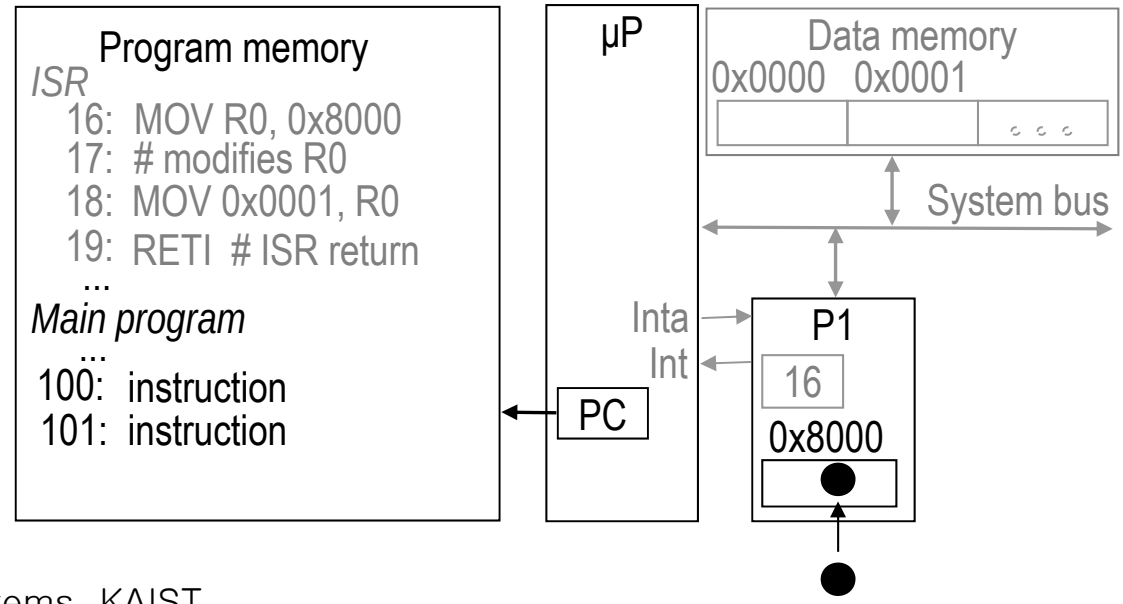
Peripheral to memory transfer *without DMA*, using vectored interrupt



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

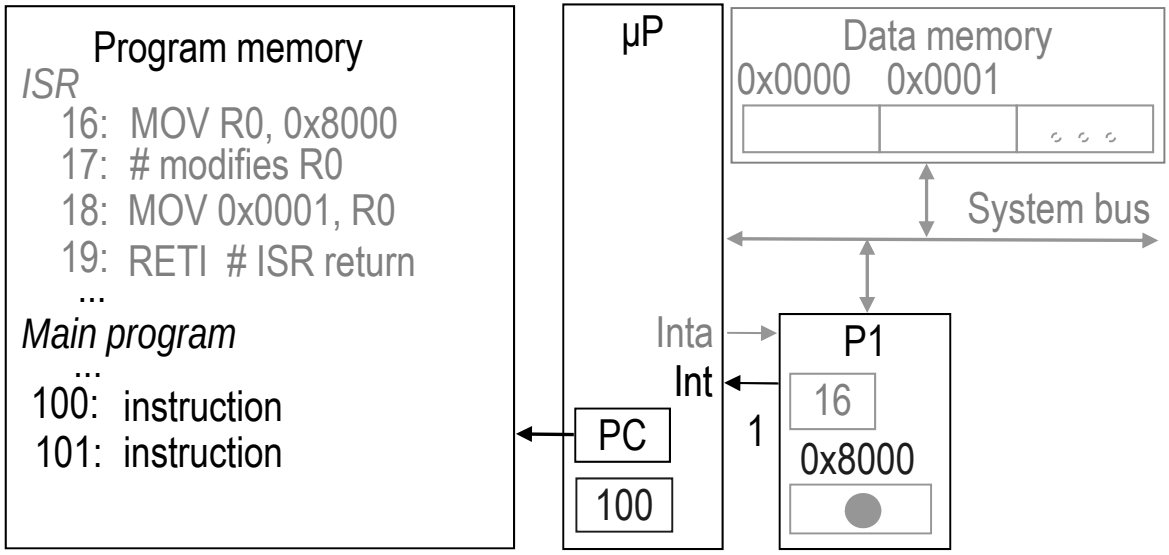
1(a): μP is executing its main program

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



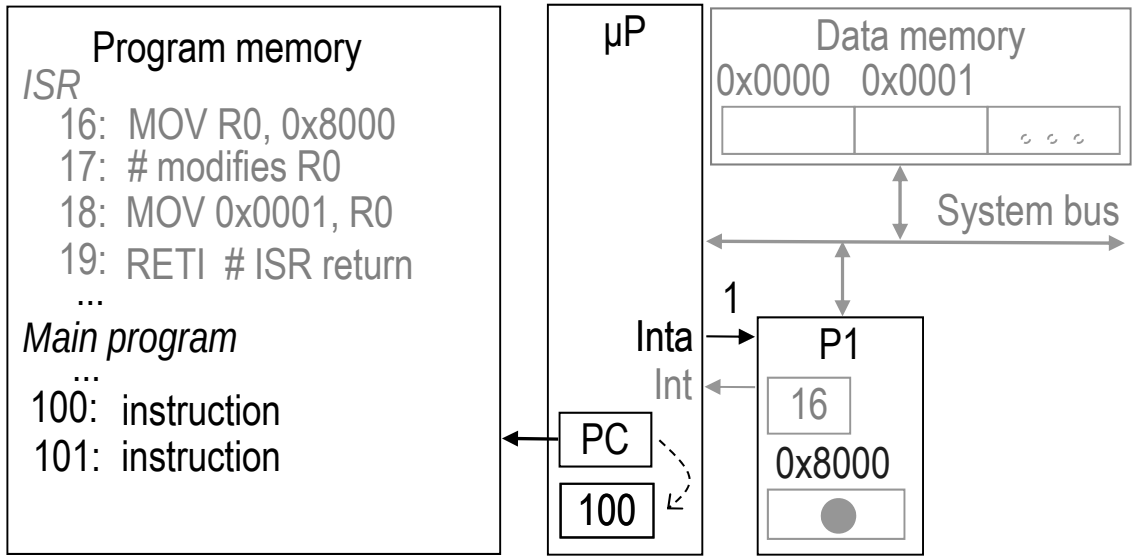
Peripheral to memory transfer *without* DMA, using vectored interrupt (II)

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



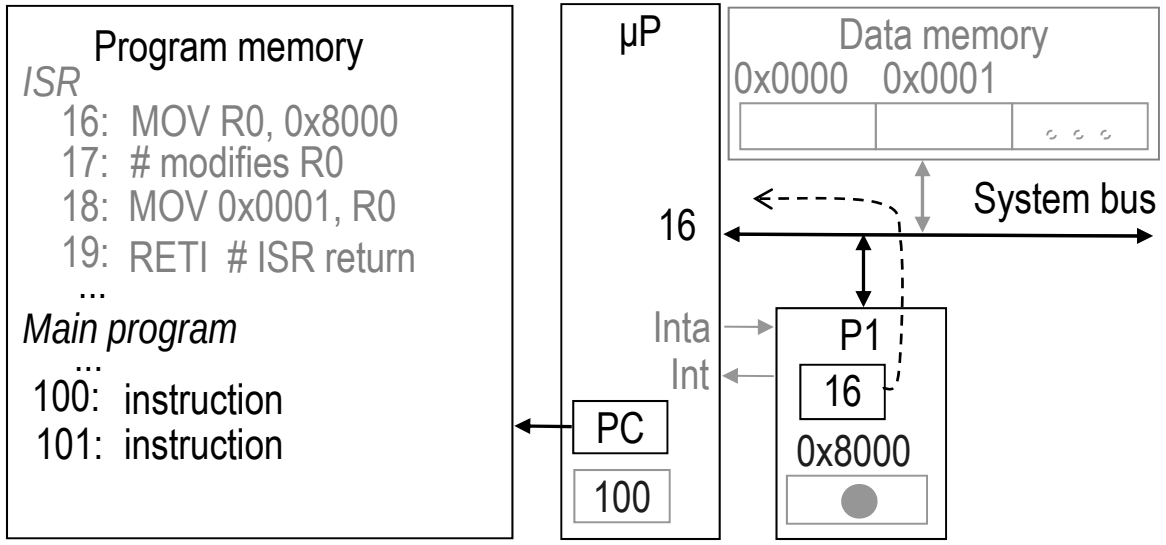
Peripheral to memory transfer *without* DMA, using vectored interrupt (III)

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 (IV)

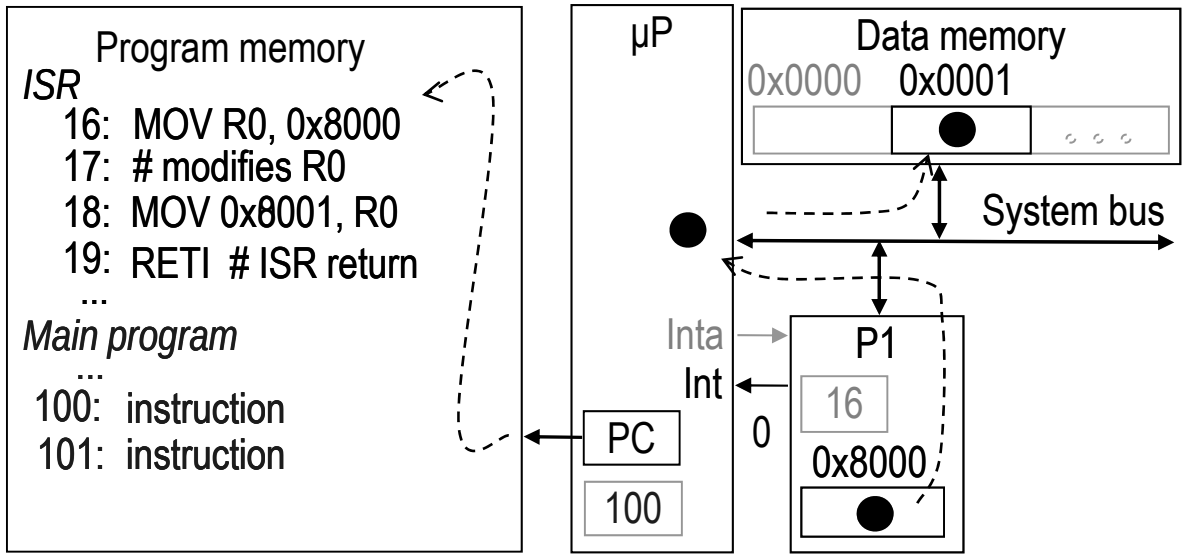
4: P1 detects *Inta* and puts interrupt address vector 16 on the data bus.



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

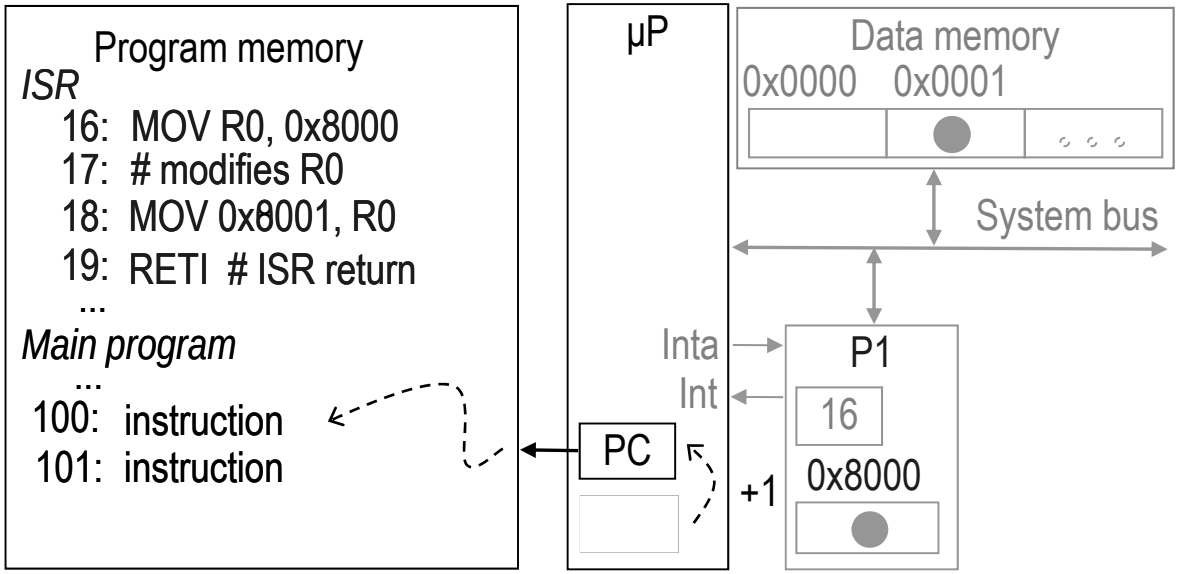
5(a): μP jumps to the address on the bus (16).
The ISR there reads data from 0x8000
and then writes it to 0x0001, which is in memory.

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

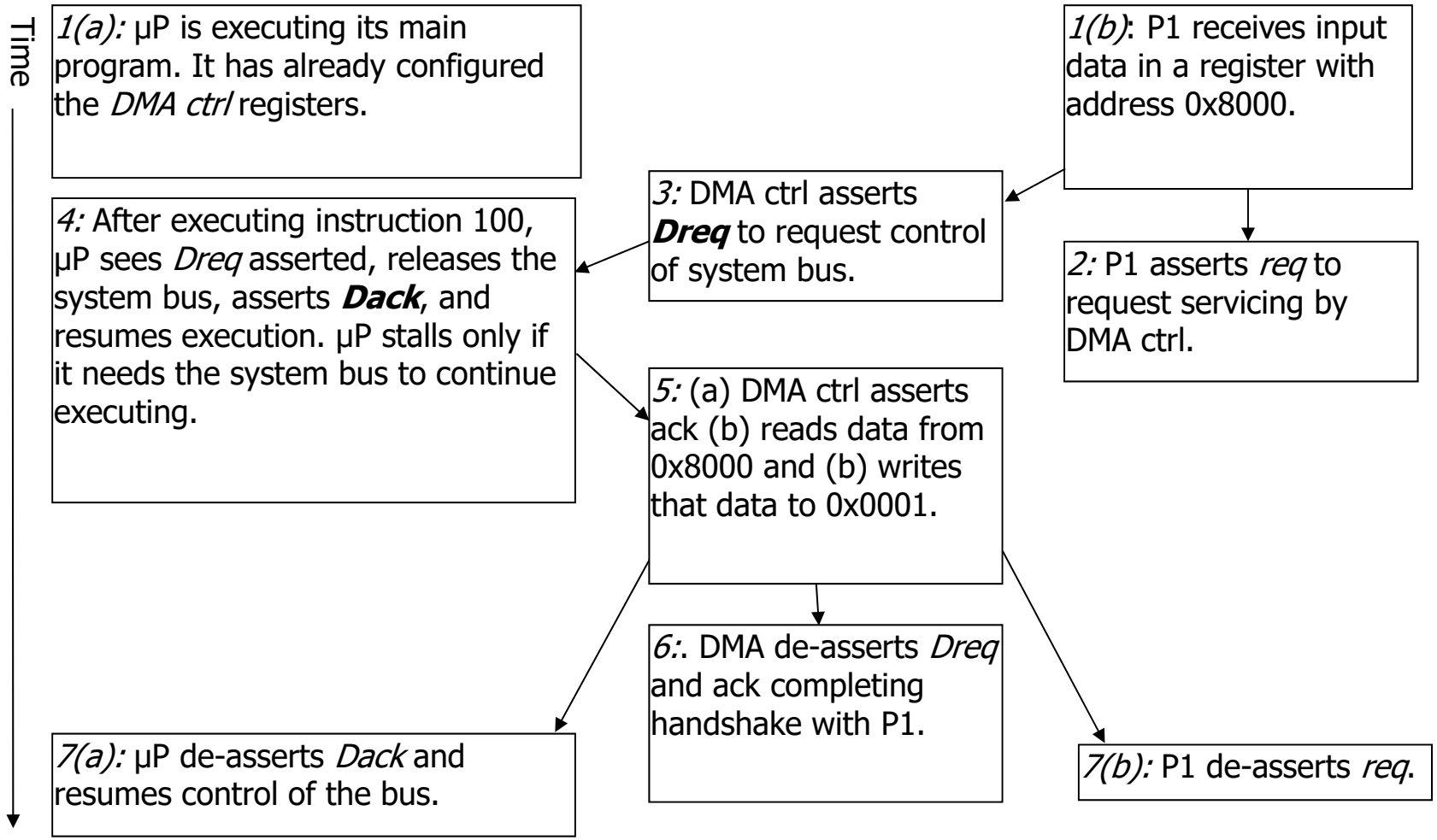


Peripheral to memory transfer *without* DMA, using vectored interrupt (VI)

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



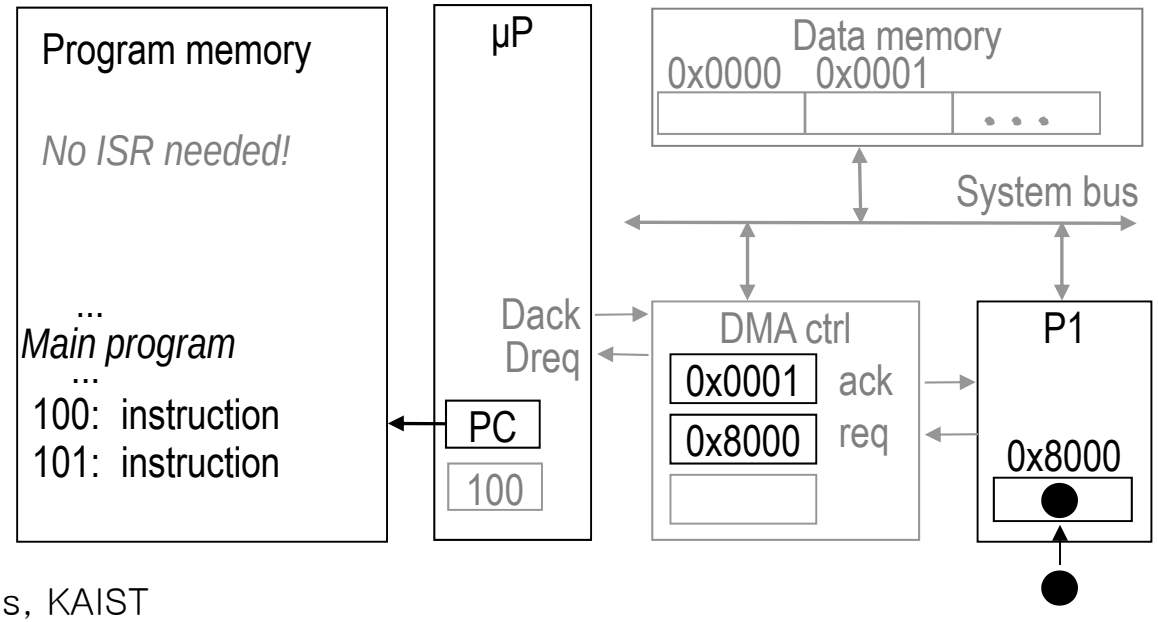
Peripheral to memory transfer *with DMA*



Peripheral to memory transfer with DMA (I)

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

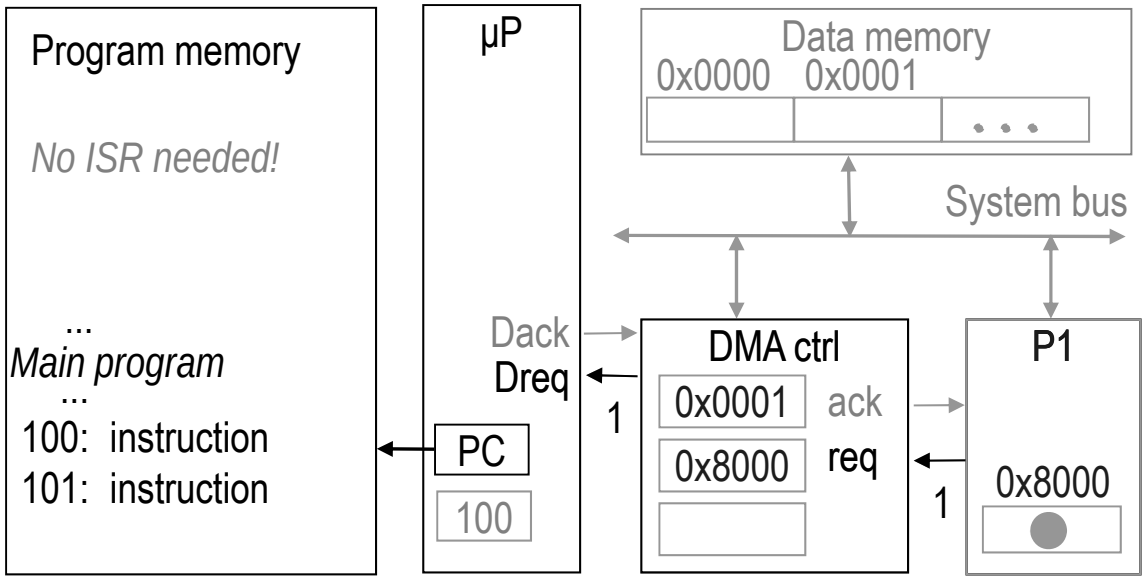
1(b): P1 receives input data in
a register with address 0x8000.



Peripheral to memory transfer with DMA (II)

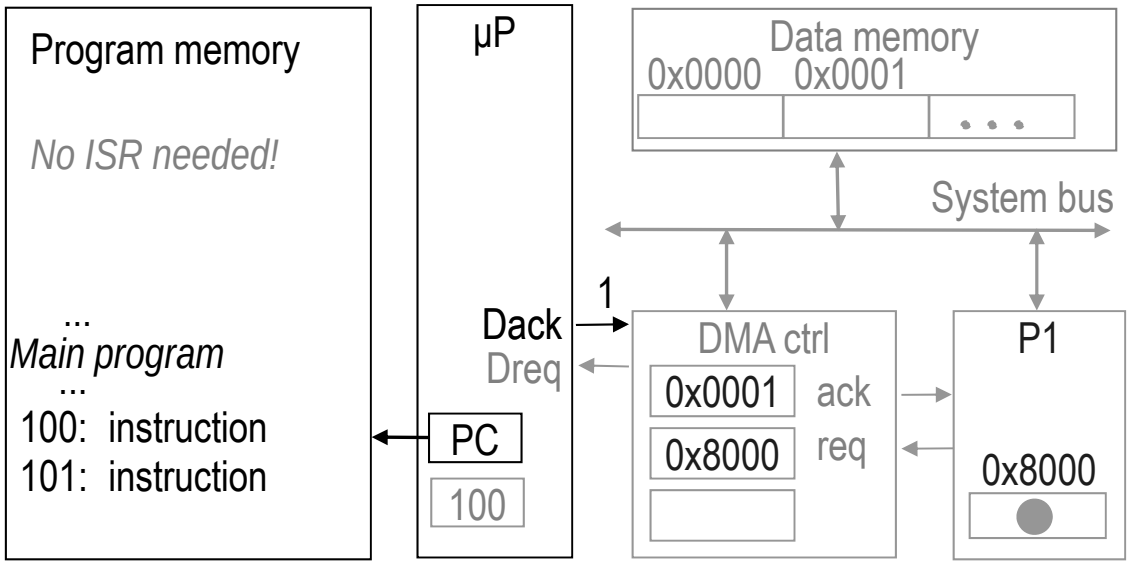
2: P1 asserts *req* to request servicing by DMA ctrl.

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



Peripheral to memory transfer with DMA (III)

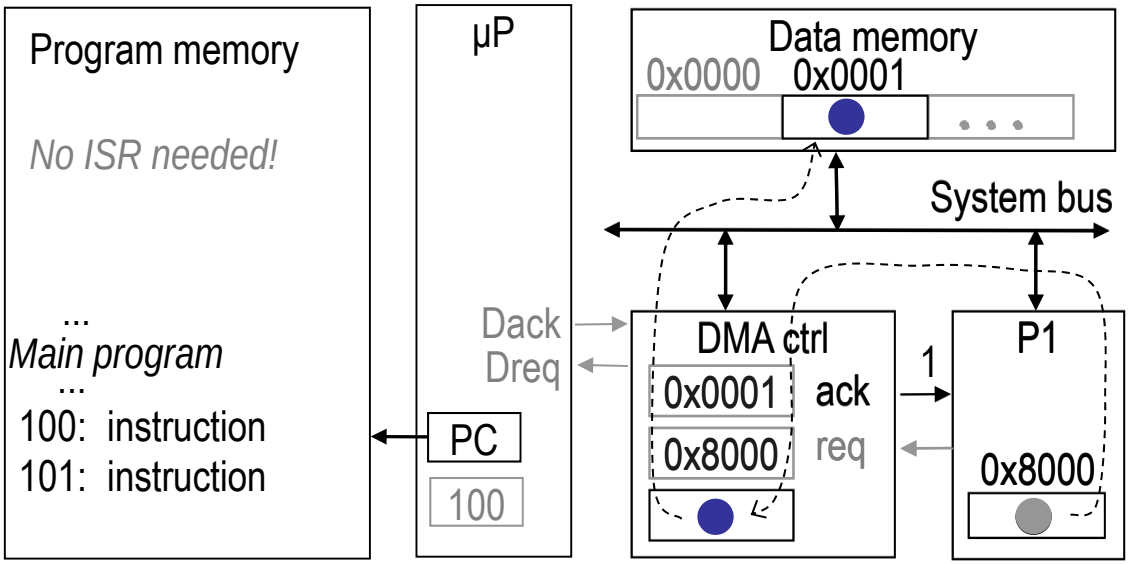
4: After executing instruction 100,
μP sees *Dreq* asserted, releases the system bus, asserts ***Dack***,
and resumes execution,
μP stalls only if it needs the system bus to continue executing.



Peripheral to memory transfer with DMA (IV)

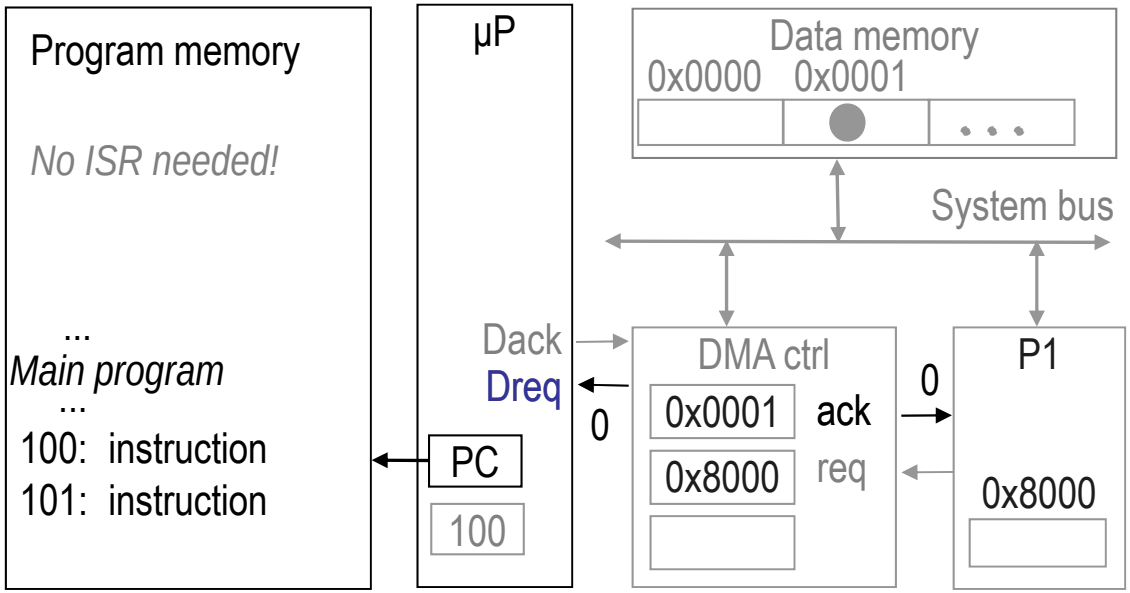
5: DMA ctrl (a) asserts ack,
(b) reads data from 0x8000, and
(c) writes that data to 0x0001.

(Meanwhile, processor still executing if not stalled!)



Peripheral to memory transfer with DMA (V)

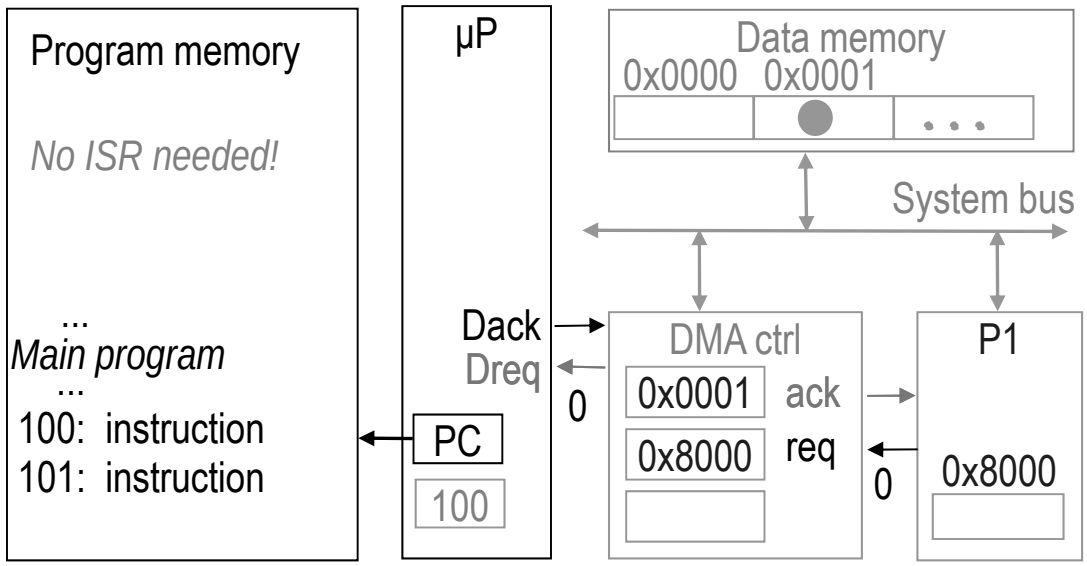
6: DMA de-asserts *Dreq* and *ack* completing the handshake with P1.



Peripheral to memory transfer with DMA (IV)

7(a): uP deasserts *Dack* and resumes control of the bus

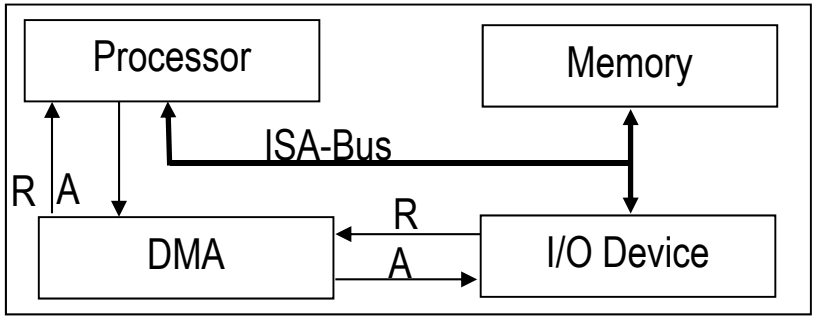
7(b): Peripheral P1 deasserts *req*.



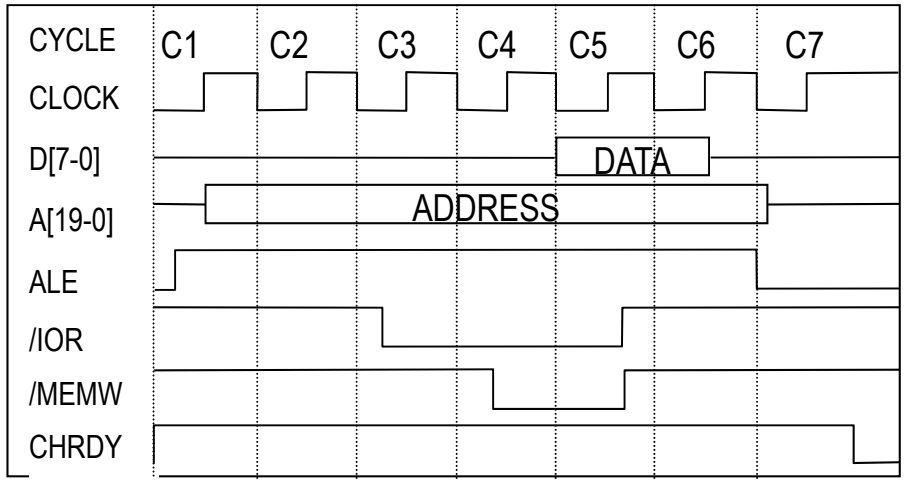
Remarks on DMA

- DMA can transfer numerous pieces of data (block)
  - E.g., 128 words for a disk controller
- We must configure the DMA controller
  - During system initialization
  - Set various DMA configuration registers
    - Single/block transfer mode
    - The number of words in a block.
- Some modern peripherals come with DMA capabilities built into the peripheral itself.

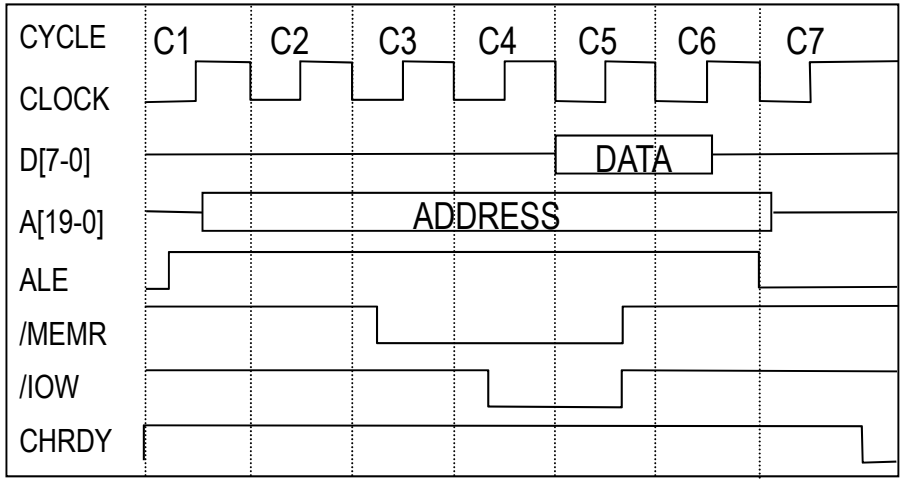
Ex: ISA Bus DMA Cycles



DMA Memory-Write Bus Cycle



DMA Memory-Read Bus Cycle

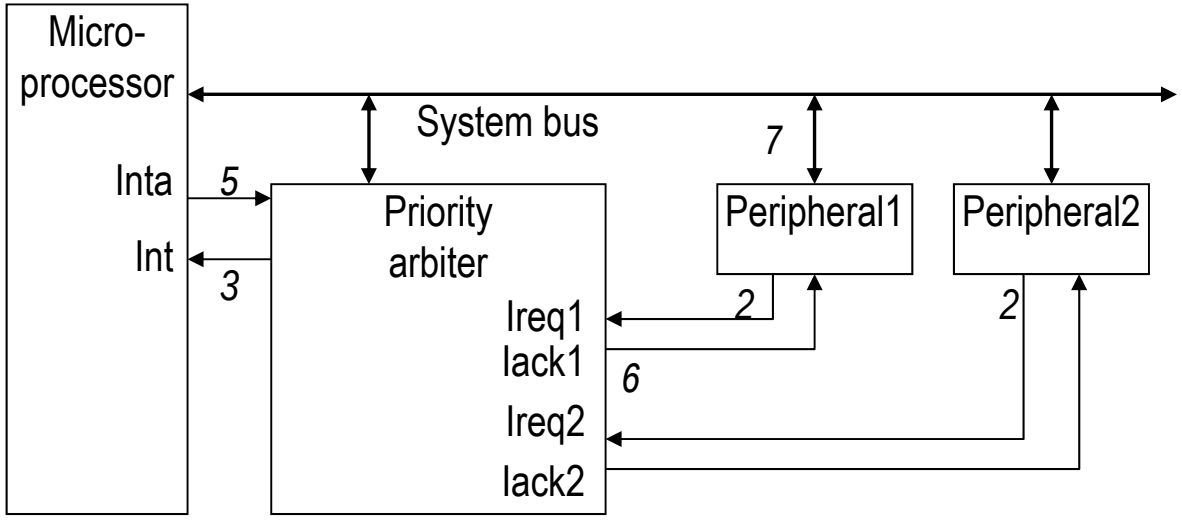


6.10 Arbitration

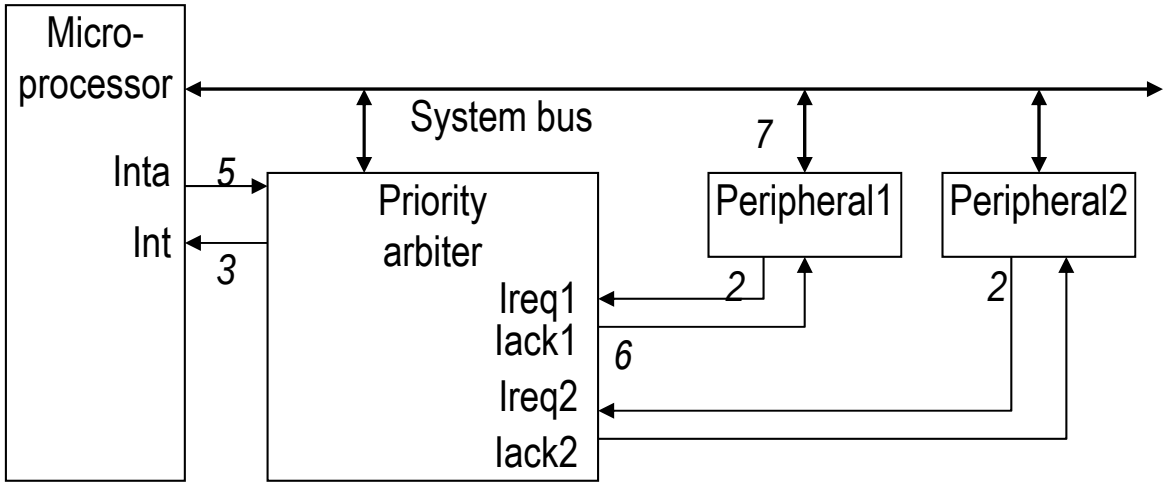
- Consider the situation where multiple peripherals request service from single resource (e.g., microprocessor, DMA controller) simultaneously - which gets serviced first?

A. 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



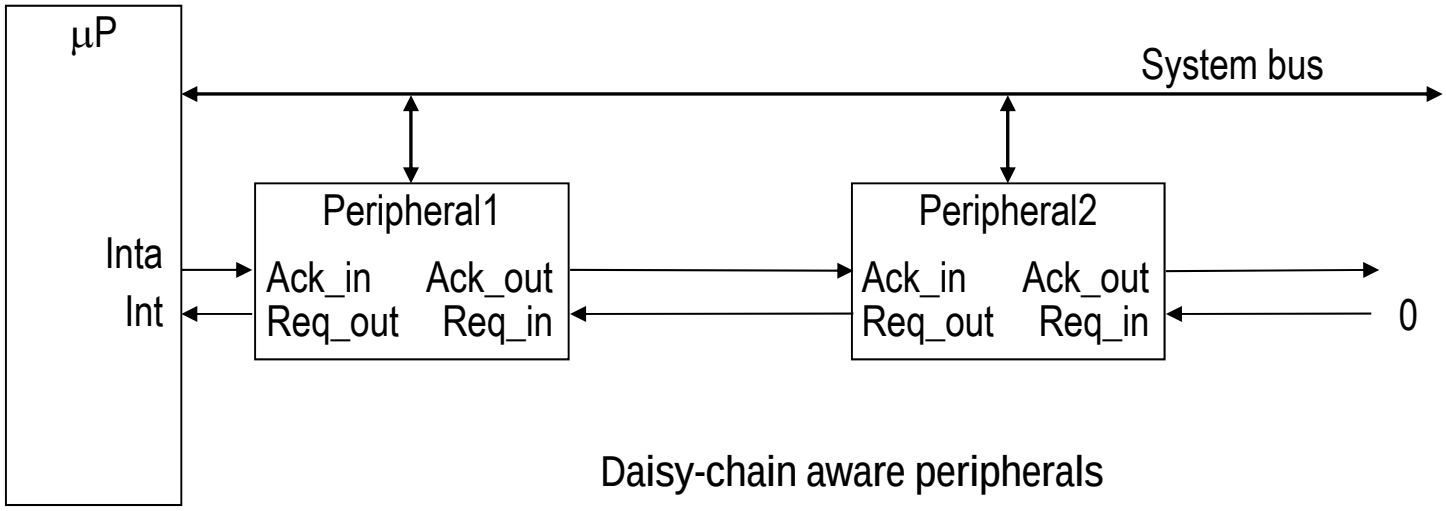
1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. Microprocessor 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
    - More equitable distribution of service.

B. Daisy-Chain Arbitration

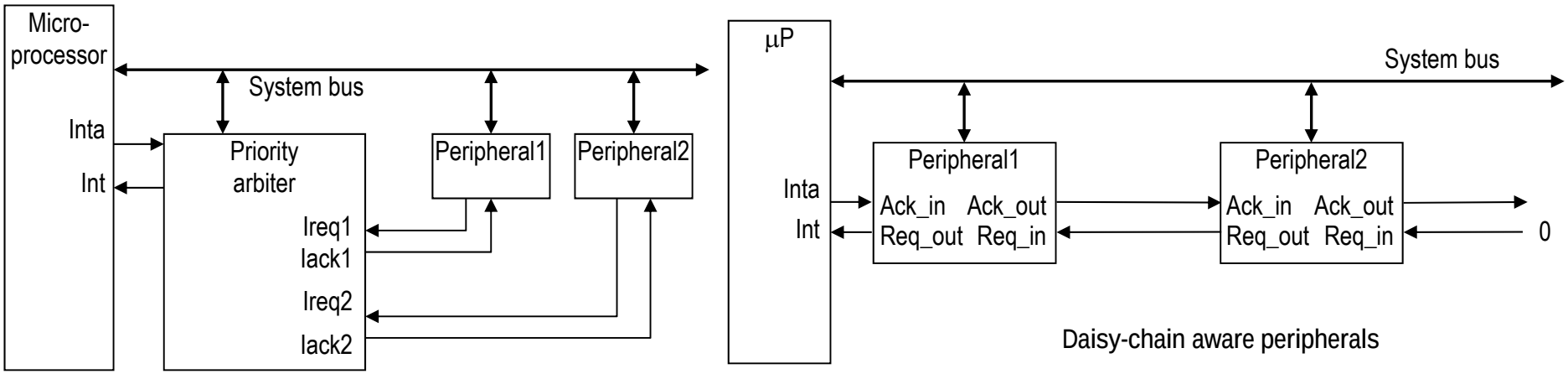
- Arbitration done by peripherals
  - Built into peripheral or external logic added
    - *req* input and *ack* output added to each peripheral
- Peripherals connected to each other in daisy-chain manner
  - One peripheral connected to resource, all others connected "upstream"
  - Peripheral's *req*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



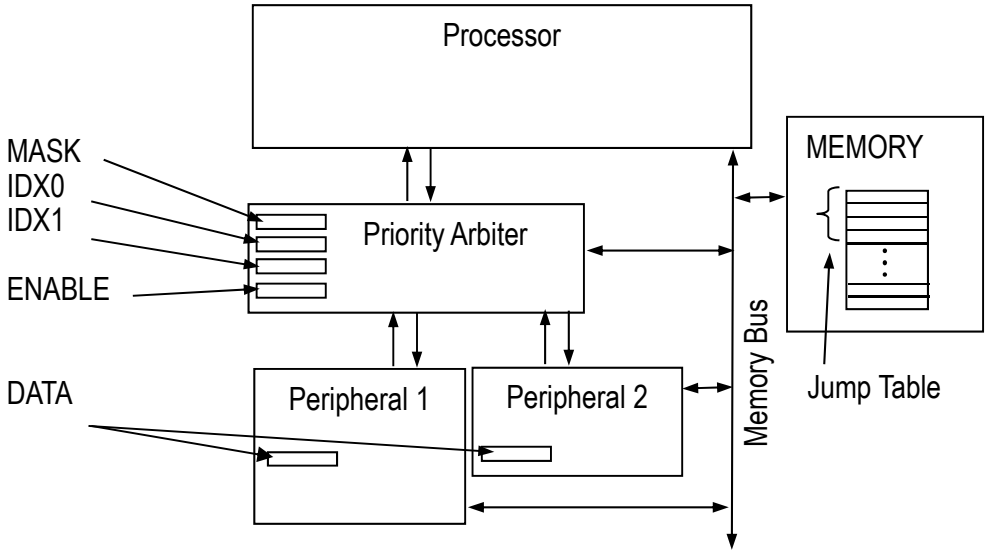
Daisy-chain aware peripherals

Daisy-Chain Arbitration (II)

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



Ex: Vectored Interrupt Using an Interrupt Table



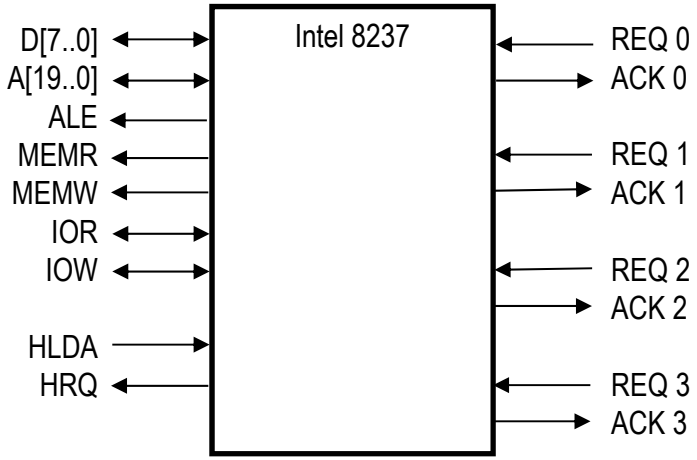
- Fixed priority: i.e., Peripheral1 has highest priority
- Keyword "_at_" followed by memory address forces compiler to place variables in specific memory locations
  - e.g., memory-mapped registers in arbiter, peripherals
- A peripheral's index into interrupt table is sent to memory-mapped register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

Ex: Vectored Interrupt Using an Interrupt Table (II)

```
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
}
```

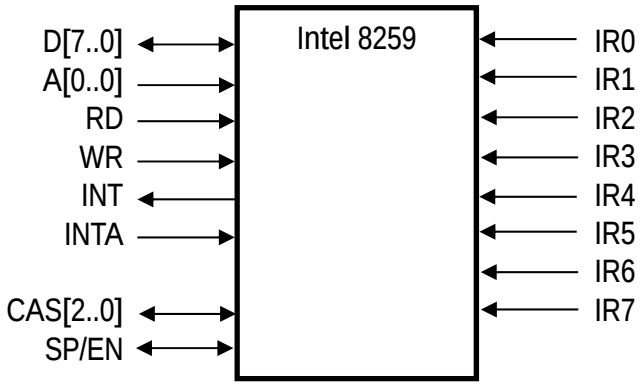
```
void Peripheral1_ISR(void) {
    unsigned char data;
    data = PERIPHERAL1_DATA_REG;
    // do something with the data
}

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;
}
```

Intel 8237 DMA Controller



Signal	Description
D[7..0]	These wires are connected to the system bus (ISA) and are used by the microprocessor to write to the internal registers of the 8237.
A[19..0]	These wires are connected to the system bus (ISA) and are used by the DMA to issue the memory location where the transferred data is to be written to. The 8237 is also addressed by the micro-processor through the lower bits of these address lines.
ALE*	This is the address latch enable signal. The 8237 use this signal when driving the system bus (ISA).
MEMR*	This is the memory write signal issued by the 8237 when driving the system bus (ISA).
MEMW*	This is the memory read signal issued by the 8237 when driving the system bus (ISA).
IOR*	This is the I/O device read signal issued by the 8237 when driving the system bus (ISA) in order to read a byte from an I/O device
IOW*	This is the I/O device write signal issued by the 8237 when driving the system bus (ISA) in order to write a byte to an I/O device.
HLDA	This signal (hold acknowledge) is asserted by the microprocessor to signal that it has relinquished the system bus (ISA).
HRQ	This signal (hold request) is asserted by the 8237 to signal to the microprocessor a request to relinquish the system bus (ISA).
REQ 0,1,2,3	An attached device to one of these channels asserts this signal to request a DMA transfer.
ACK 0,1,2,3	The 8237 asserts this signal to grant a DMA transfer to an attached device to one of these channels.
*See the ISA bus description in this chapter for complete details.	

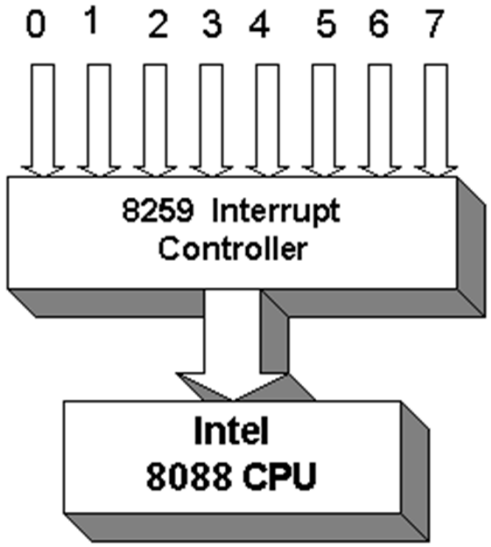
Intel 8259 Programm-able Priority Controller



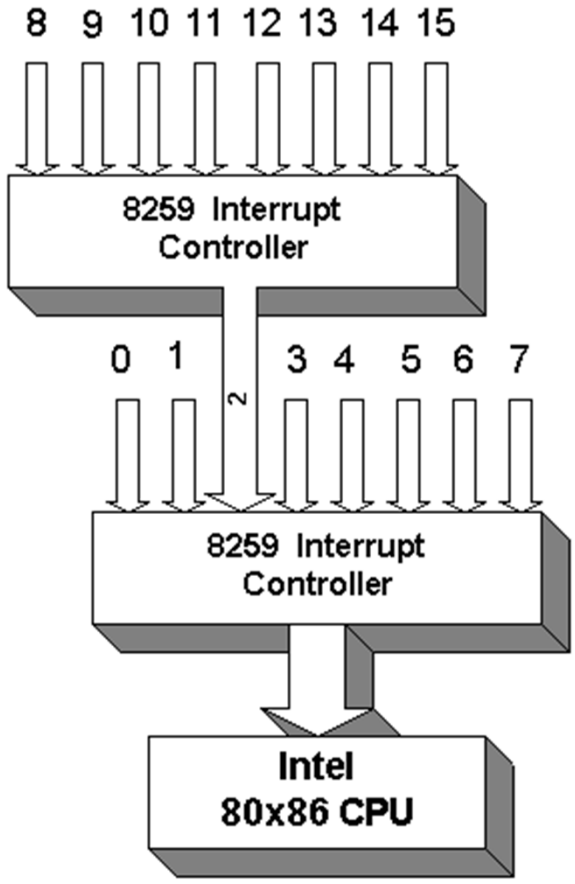
Signal	Description
D[7..0]	These wires are connected to the system bus and are used by the microprocessor to write or read the internal registers of the 8259.
A[0..0]	This pin actis in cunjunction with WR/RD signals. It is used by the 8259 to decipher various command words the microprocessor writes and status the microprocessor wishes to read.
WR	When this write signal is asserted, the 8259 accepts the command on the data line, i.e., the microprocessor writes to the 8259 by placing a command on the data lines and asserting this signal.
RD	When this read signal is asserted, the 8259 provides on the data lines its status, i.e., the microprocessor reads the status of the 8259 by asserting this signal and reading the data lines.
INT	This signal is asserted whenever a valid interrupt request is received by the 8259, i.e., it is used to interrupt the microprocessor.
INTA	This signal, is used to enable 8259 interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the microprocessor.
IR 0,1,2,3,4,5,6,7	An interrupt request is executed by a peripheral device when one of these signals is asserted.
CAS[2..0]	These are cascade signals to enable multiple 8259 chips to be chained together.
SP/EN	This function is used in conjunction with the CAS signals for cascading purposes.

IBM PC Interrupt Control

- IBM PC XT
  - 8088



- IBM PC AT
  - 80286

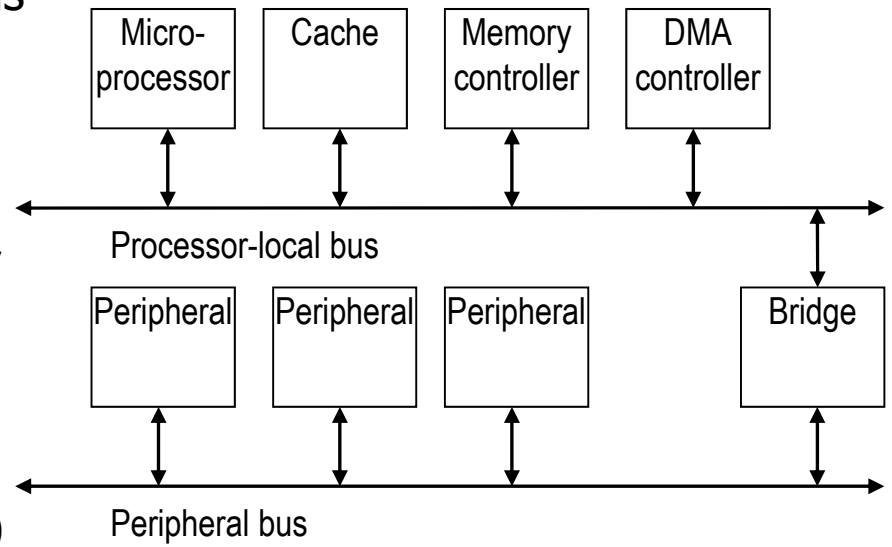


(C) http://www.windowsnetworking.com/articles_tutorials/interupt.html

6.11 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.

References

- [1] Frank Vahid, “Embedded system design: A unified hardware/software introduction”, John Wiley & Sons, 2002.