ECE 1175
Embedded Systems Design

Midterm Review

Wei Gao

Online Midterm Exam

- **When**: next Tuesday (10/19) in class
- 25% of your final grade
- **What about**: Everything from the beginning of class
  - Closed-book, closed-notes
  - All included in lecture notes, lab and homework assignments
  - May have some programming questions
  - Make your answers short to include only key points
  - Sample questions will be posted on the course website

Definition

- Embedded system: any device that includes a computer but is not itself a general-purpose computer.

- Application specific
  - The design is specialized and optimized for specific application
  - Don’t need all the general-purpose bells and whistles

System Characteristics

- Non-functional requirements
  - Real-time
  - Low power
  - Small memory footprint
  - Low cost
- Short development cycles
- Small teams

Real-Time

- Hard real time: violating timing constraints causes failure
  - Anti-lock Brake System
  - CD burner
  - Software modem
- Soft real time: missing deadline results in degraded performance
  - Online video
  - GPS map
  - Audio (MP3 player)?

Alternative Technology

- Application-Specific Integrated Circuits (ASICs)
- Microprocessors
- Field-Programmable Gate Arrays (FPGAs)

- Why should we use microprocessors?

ASIC

- Ex: Digital baseband processing for cell phones
- Pros
  - Performance: Fast!
  - Power: Fewer logic elements lead to low power
- Cons
  - Development cost: Very high
    - 2 million $ for starting production of a new ASIC
    - Needs a long time and a large team
  - Reprogrammability: None!
    - Single-purpose devices
    - Difficult to upgrade systems

Microprocessors

- Performance
  - Con: Programmable architecture is fundamentally slow!
    - Fetch, decode instructions
  - Pro: Highly optimized architecture and manufacturing
    - Pipelines; cache; clock frequency; circuit density; manufacturing technology
- Power
  - Processors perform poorly in terms of performance/watt!
  - Power management can alleviate the power problem.
- Flexibility, development cost and time
  - Let software do the work!

FPGA

- Programmable hardware
- In the middle of ASIC and microprocessor
  - Power
    - Hardware implementation: better performance/watt than microprocessor
    - Many overhead transistors: waste more power than ASIC
  - Reprogrammability
    - Reprogammable: lower development cost than ASIC
    - More difficult to program than microprocessor
- More commonly used for prototyping

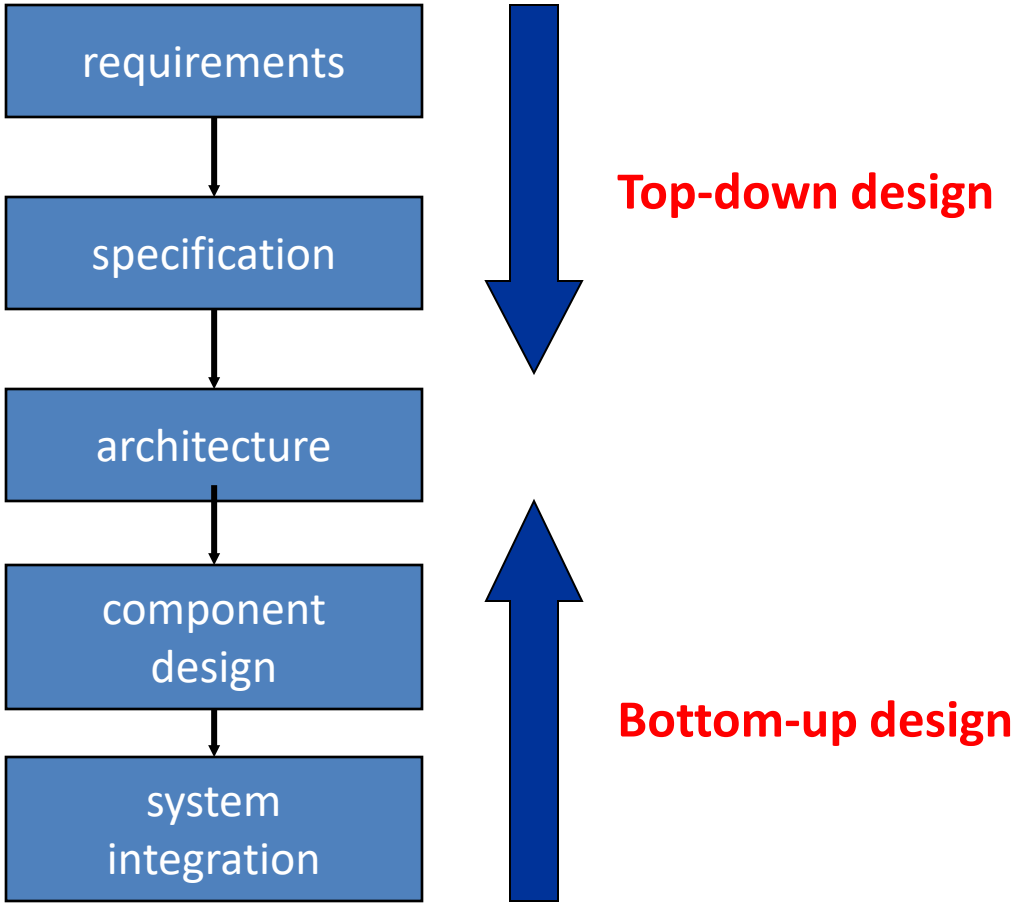
State of the Practice

- Microprocessor is the dominant player
  - Reprogrammability and low development cost >> low performance/watt
  - Optimization and power management have significantly improved microprocessors' performance
- Microprocessor + ASIC is common
  - Ex: cell phone
- FPGA is expected to improve

Design Challenges

- Non-functional constraints
  - How do we meet deadlines?
    - Faster hardware or better software?
  - How do we minimize power?
    - Turn off unnecessary logic? Reduce memory accesses? Slow down CPU?
  - Cost considerations
- Trade-offs among constraints
- Optimization & analysis are important!

Design Methodologies



Microprocessors

- von Neumann
  - Same memory holds data, instructions.
  - A single set of address/data buses between CPU and memory
- Harvard
  - Separate memories for data and instructions.
  - Two sets of address/data buses between CPU and memory

von Neumann vs. Harvard

- Harvard allows two simultaneous memory fetches.
- Harvard architectures are widely used because
  - Most DSPs use Harvard for streaming data
  - The separation of program and data memories
    - Greater memory bandwidth
    - Higher performance for digital signal processing
  - Speed is gained at the expense of more complex electrical circuitry.
- Other examples: On chip cache of CPUs is divided into an instruction cache and a data cache

RISC vs. CISC

- Reduced Instruction Set Computer (RISC)
  - Compact, uniform instructions: facilitate pipelining
  - More lines of code: poor memory footprint
  - Allow effective compiler optimization
- Complex Instruction Set Computer (CISC)
  - Many addressing modes and instructions;
  - High code density.
  - Often require manual optimization of assembly code for embedded systems.

Busy-Wait I/O Programming

- Simplest way to program I/O devices.
  - Devices are usually slower than CPU and require more cycles
  - CPU has to wait for device to finish before starting next one
  - Use **`peek`** instruction to test when device is finished

```
//send a string to device using Busy-Wait handshaking
current_char = mystring;
while (*current_char != '\0') {
    //send character to device (data register)
    poke(OUT_CHAR,*current_char);
    //wait for device to finish by checking its status
    while (peek(OUT_STATUS) != 0);
    //advance character pointer to next one
    current_char++;
}
```

Mutual Exclusion using Busy-Wait

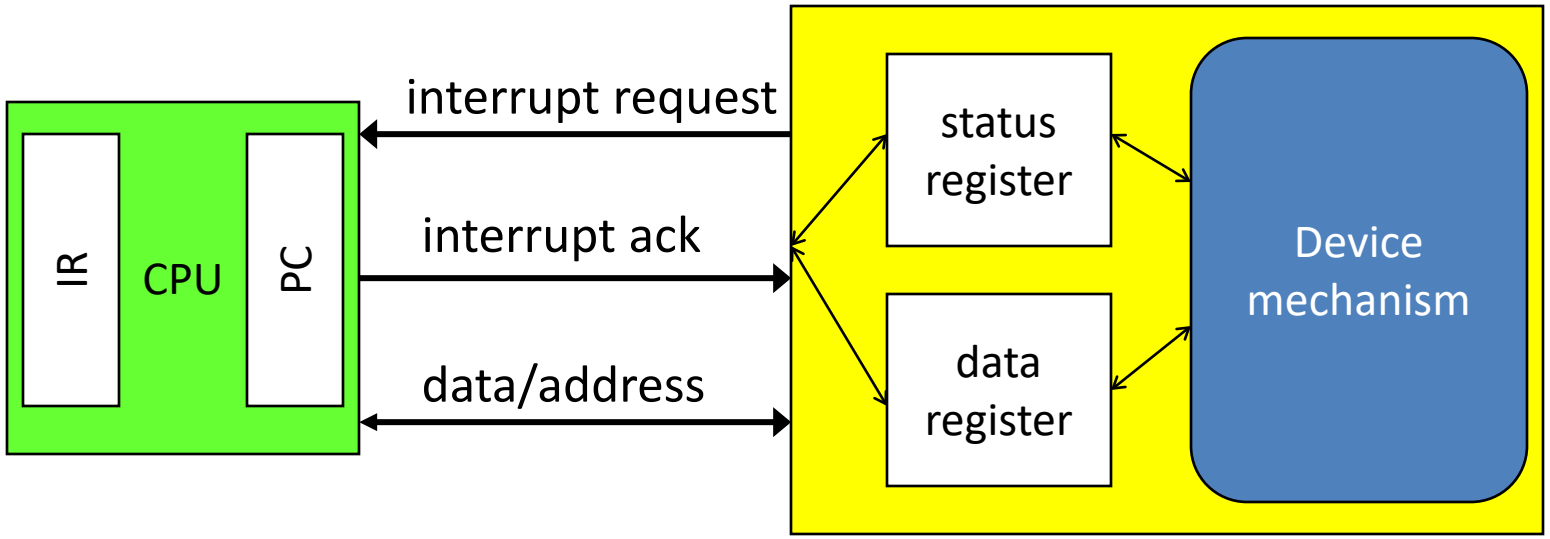
- Exclusive access on the I/O device
  - Multi-threaded systems

```
TestAndSet() {
    oldValue=peek(LOCK); //read
    poke(LOCK, true); //write
    return oldValue;
}
.
.
//wait until LOCK is acquired
while (TestAndSet());
//send character to device (data register)
poke(OUT_CHAR,*current_char);
//wait for device to finish by checking its status
while (peek(OUT_STATUS) != 0);
//advance character pointer to next one
current_char++;
poke(LOCK, false); //release lock
```

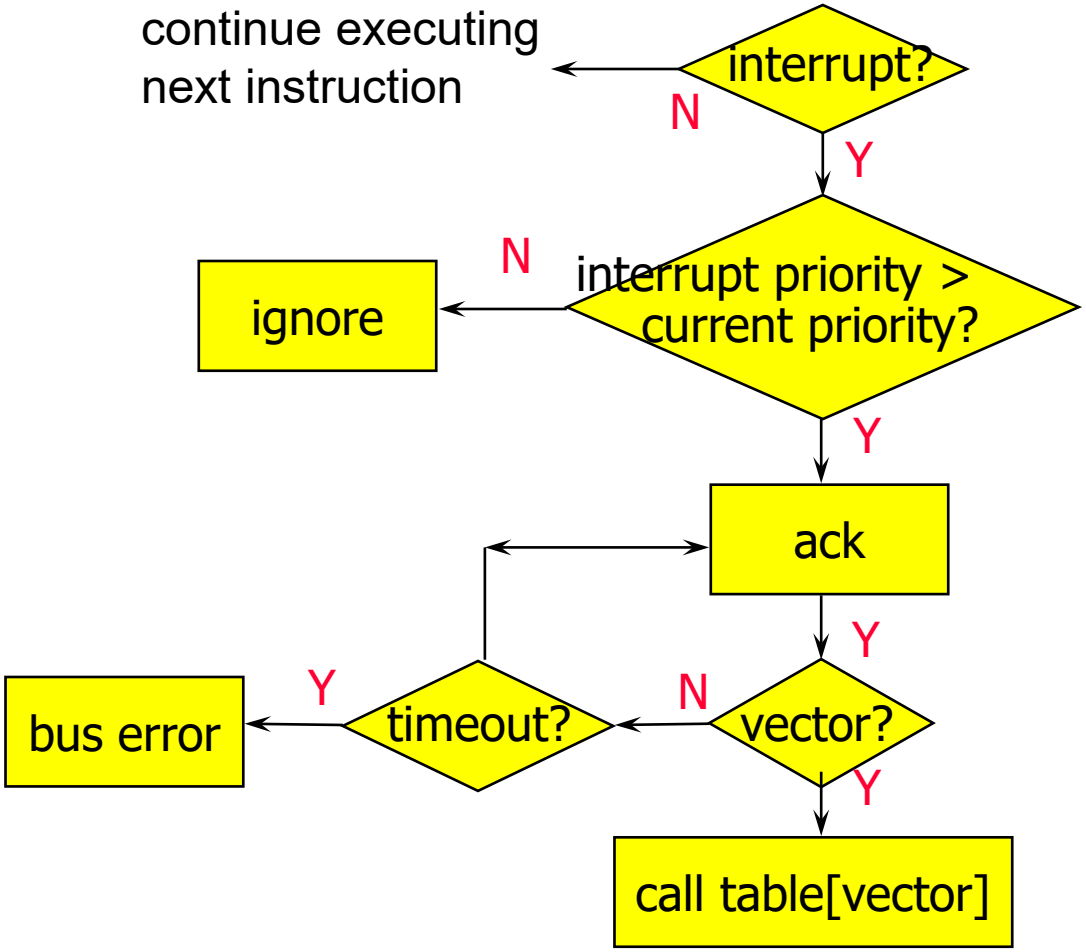
"TestAndSet" for mutual exclusion

Interrupt-based I/O

- Busy-wait is very inefficient.
  - CPU can't do other work while testing device.
  - Hard to do simultaneous I/O.
- Interrupts allow to change the flow of control in the CPU.
  - Call interrupt handler (i.e. device driver) to handle device.

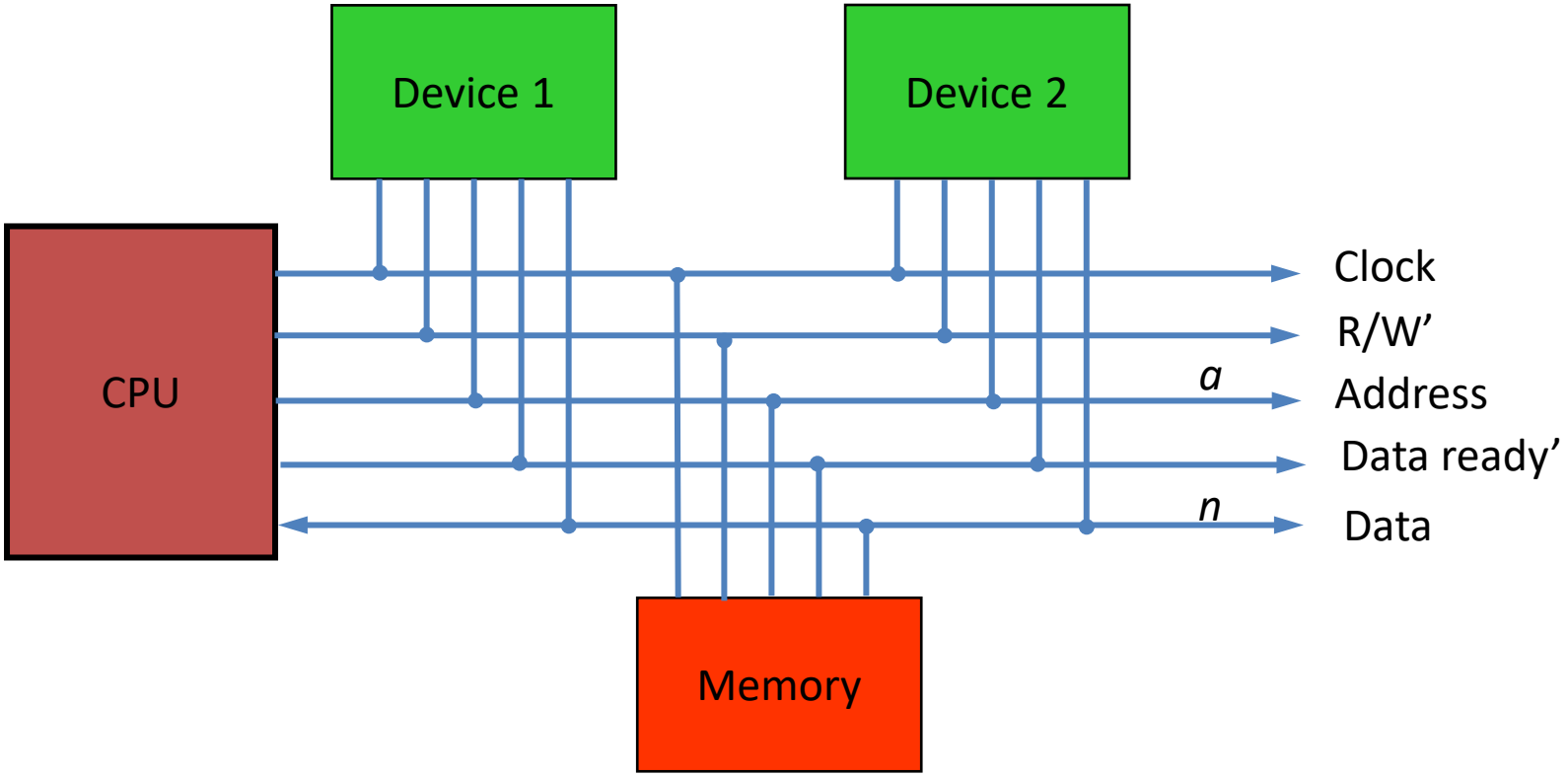


Generic Interrupt Mechanism



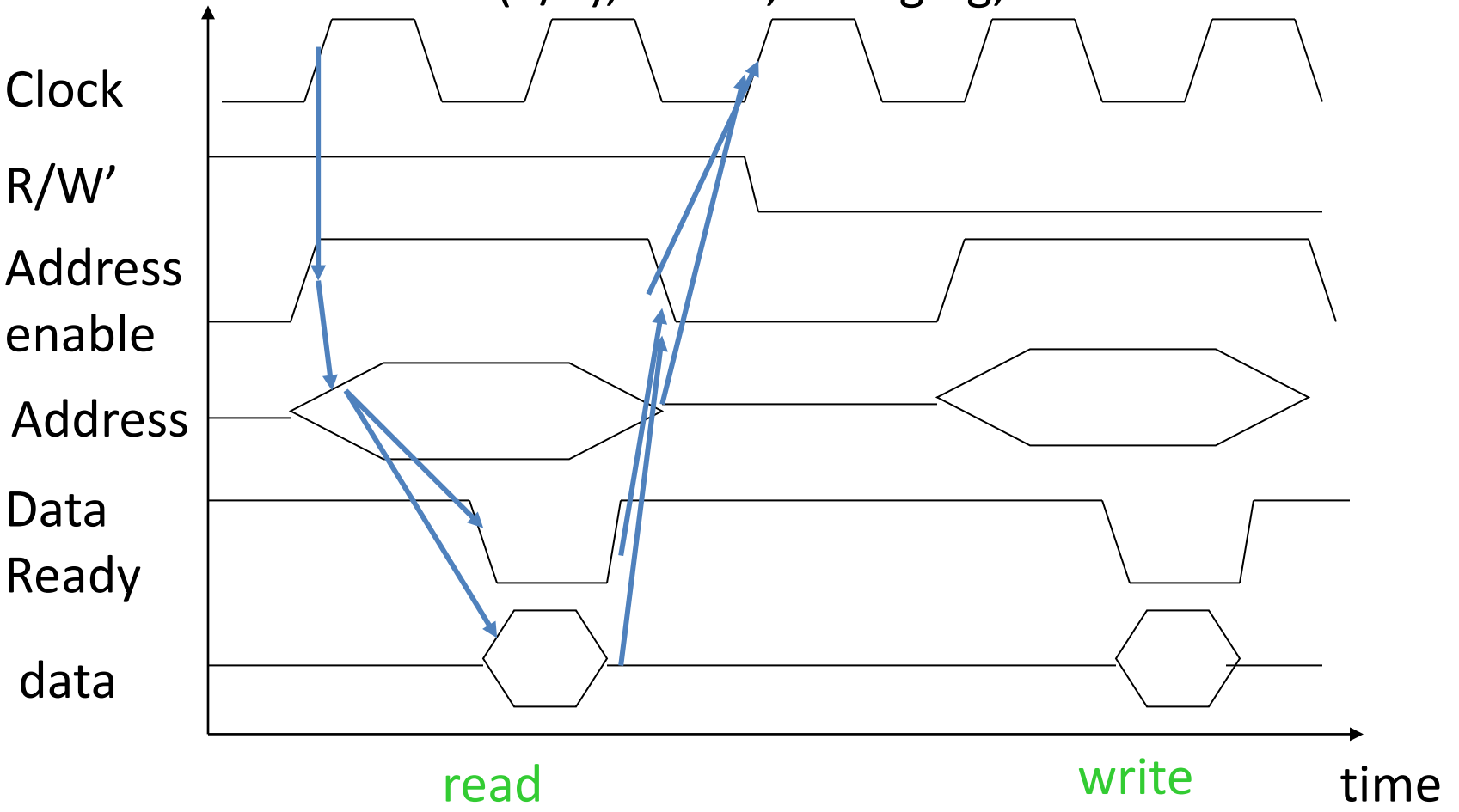
Microprocessor Bus

- Bus is a set of wires and a protocol for the CPU to communicate with memory and devices
- Five major components to support reads and writes



Typical Bus Access

- Timing diagram syntax:
  - Constant value (0/1), stable, changing, unknown.



I/O Interfaces

- Parallel I/O and Serial I/O
  - Parallel I/O: multiple input/output simultaneously
    - Data Bus, Address Bus, Intel 8255, printer
  - Serial I/O: transferring data between CPU and peripherals one bit at a time, sequentially
    - Ethernet, USB, Inter-integrated Circuit, Serial Peripheral Interface

I/O Interfaces

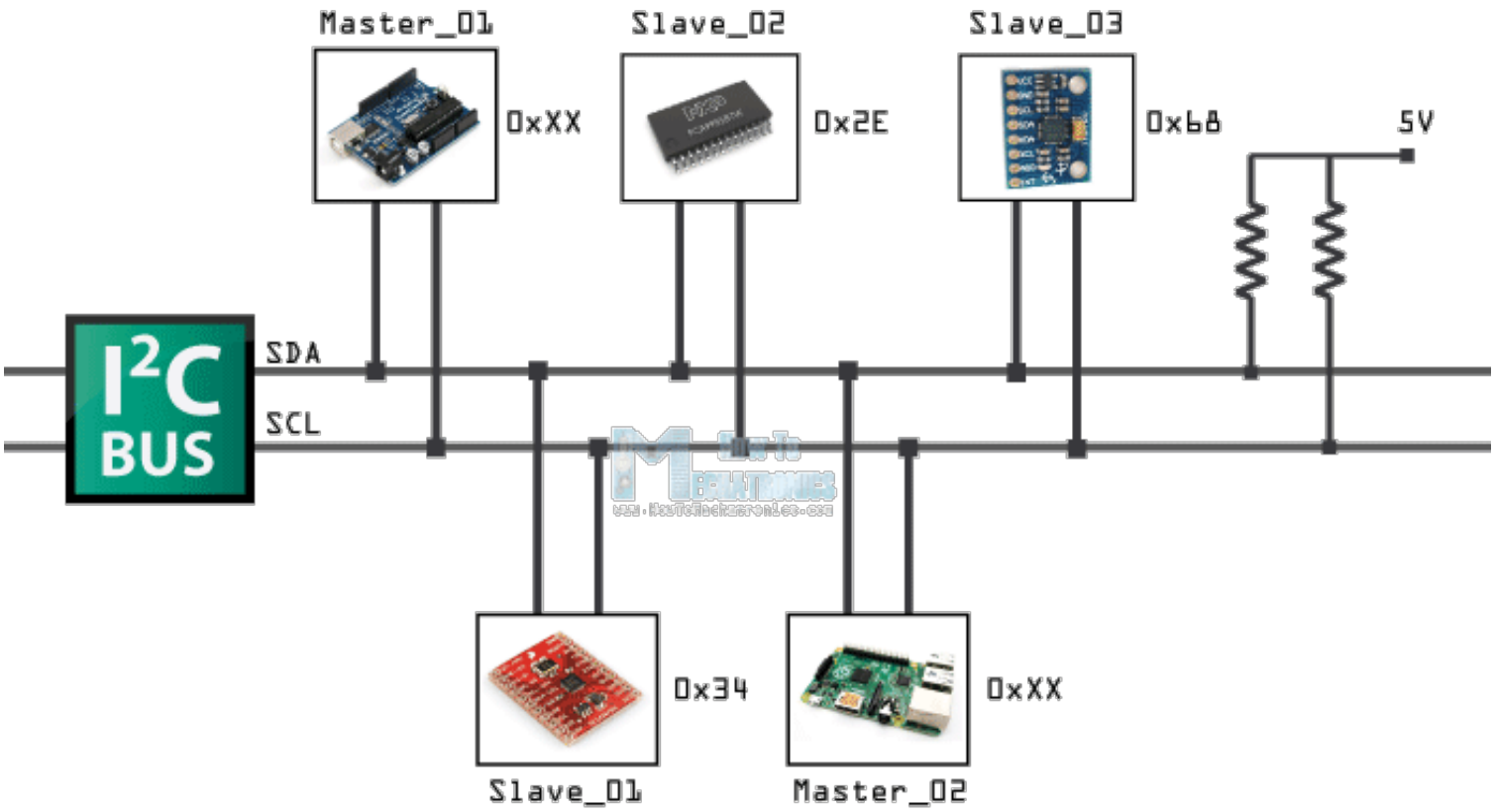
- Parallel v.s. Serial
  - Parallel
    - Wider bandwidth
    - More wires indicate more overhead
    - Simple I/O operation
  - Serial
    - 1-bit transfer per time unit
    - Less wires indicate less overhead
    - Complex I/O protocol

I/O Interfaces

- Serial over Parallel
  - Parallel interfaces have less reliability
    - Interference and noise corrupt data
    - Capacitance and mutual inductance affects bandwidth
  - Serial
    - Less mutual interference between wires
    - Higher clock frequency increases transmission rate

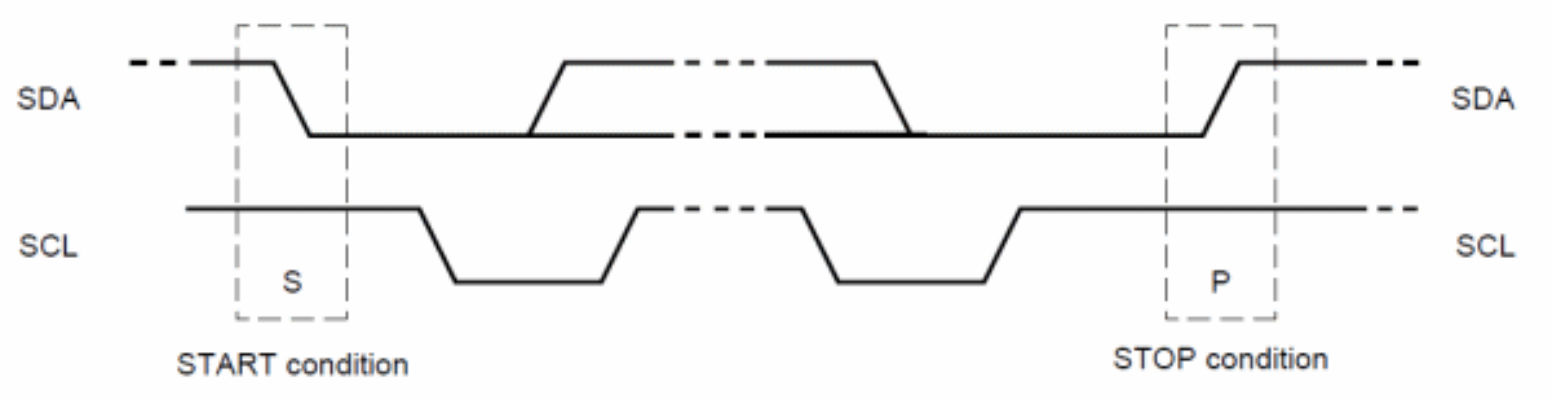
Inter-integrated Circuit (I2C)

- I^2C connection example



Inter-integrated Circuit (I2C)

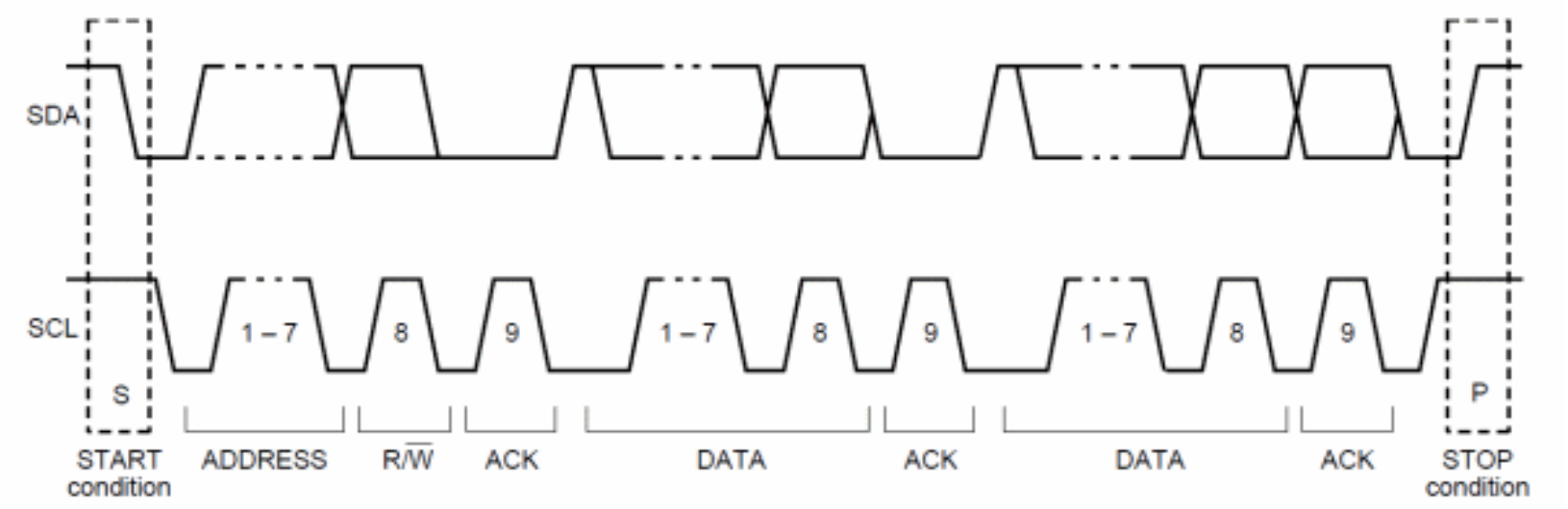
- Start and Stop condition
  - Both initiated by master
  - SCL has to be high in both case



  - SDA
    - High to low: START
    - Low to high: STOP

Inter-integrated Circuit (I2C)

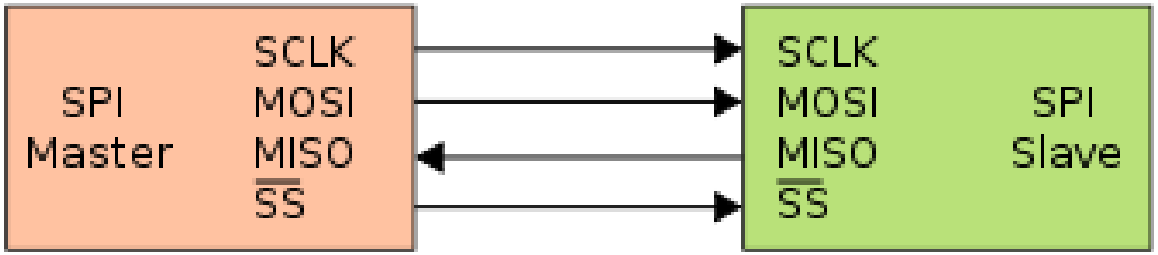
- Communication with 7-bit I2C Address



  - Initiating communication
  - Addressing slave device
  - Transferring data
  - Ending communication

Serial Peripheral Interface (SPI)

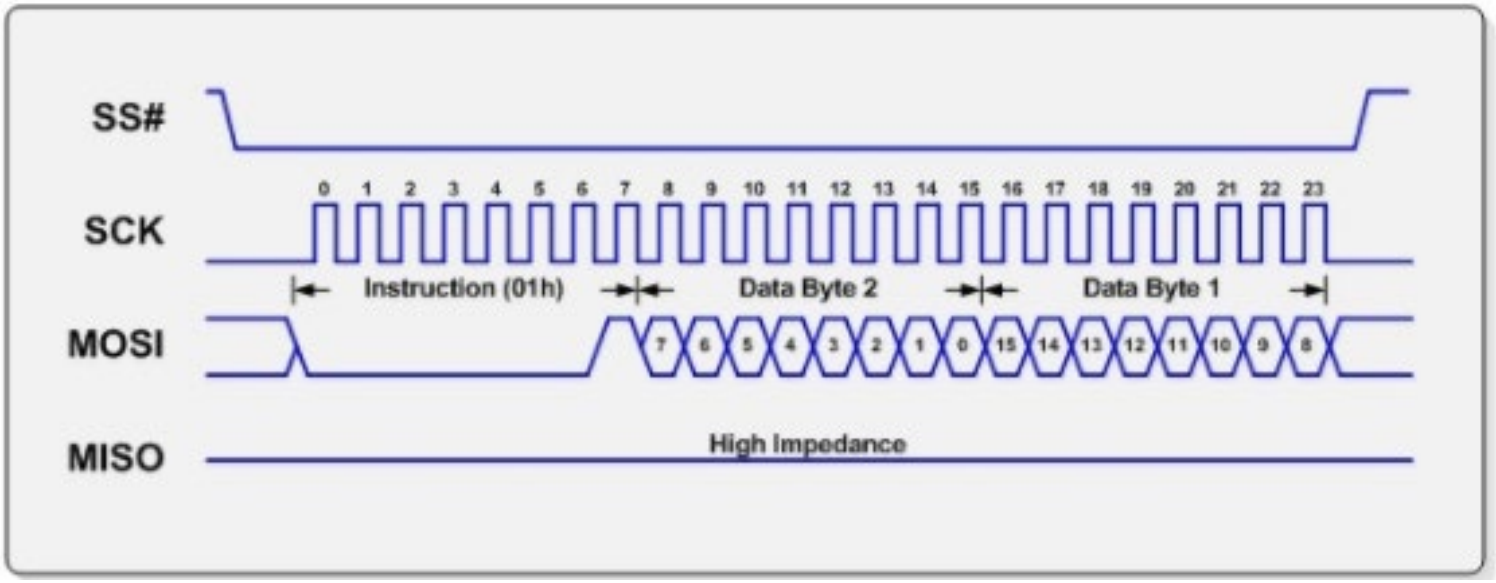
- Basic Model



  - Serial Click (SCLK or SCK): clock pulse that synchronizes data transmission generated by master
  - Master In Slave Out (MISO): slave line for sending data to master
  - Master Out Slave In (MOSI): master line for sending data to peripherals.
  - Slave Select(SS): pin on which device the master could use to enable/disable specific devices

Serial Peripheral Interface

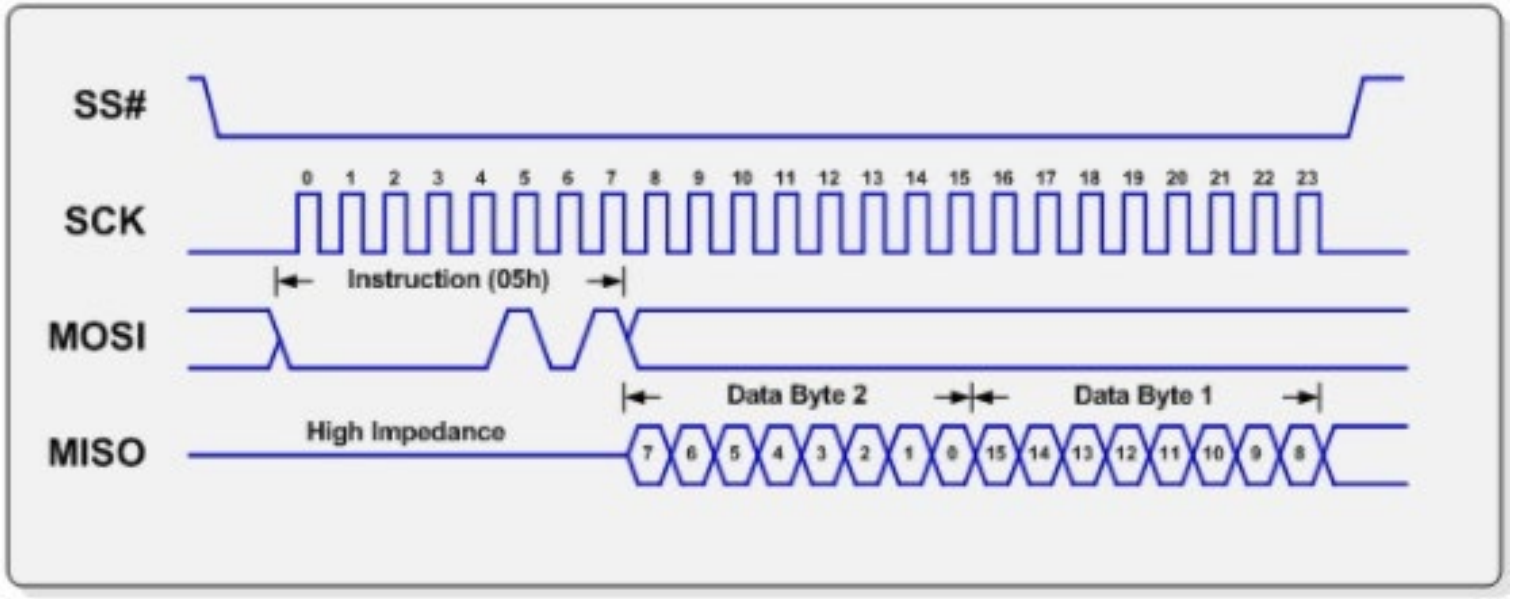
- Write/Read Transaction
  - Write Transaction



- lower SS# to select slave device
- sending instruction bytes and data bytes via MOSI

Serial Peripheral Interface

- Write/Read Transaction
  - Read Transaction



- lower SS# to select slave device
- sending instruction byte via MOSI and receiving data byte by MISO

SPI v.s. I2C

- Which one?
  - I2C require two wires while SPI may need more
  - SPI support full-duplex communication while I2C is slower
  - I2C is more power-consuming than SPI
  - I2C has ACK to verify data transfer while SPI is not
  - I2C may have multiple master but SPI only has one master

ARM Architecture

- Dominant architecture for embedded systems

ARM Instruction Sets

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
 CMP   r3,#0
  BEQ   skip
  ADD   r0,r1,r2
skip
```

```
CMP   r3,#0
ADDNE r0,r1,r2
```

Condition Codes

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	**Z=1**
NE	Not equal	**Z=0**
CS/HS	Unsigned higher or same	**C=1**
CC/LO	Unsigned lower	**C=0**
MI	Minus	**N=1**
PL	Positive or Zero	**N=0**
VS	Overflow	**V=1**
VC	No overflow	**V=0**
HI	Unsigned higher	**C=1 & Z=0**
LS	Unsigned lower or same	**C=0 or Z=1**
GE	Greater or equal	**N=V**
LT	Less than	**N!=V**
GT	Greater than	**Z=0 & N=V**
LE	Less than or equal	**Z=1 or N=!V**
AL	Always	

Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP       r0,#0
    MOVEQ     r0,#1
    BLEQ      func
```

- Set the flags, then use various condition codes

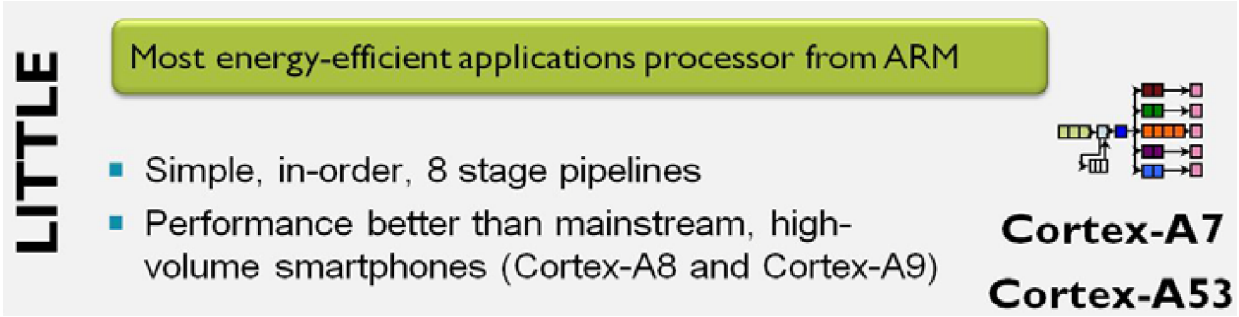
```
if (a==0) x=0;
if (a>0)  x=1;
    CMP       r0,#0
    MOVEQ     r1,#0
    MOVGT     r1,#1
```

- Use conditional compare instructions

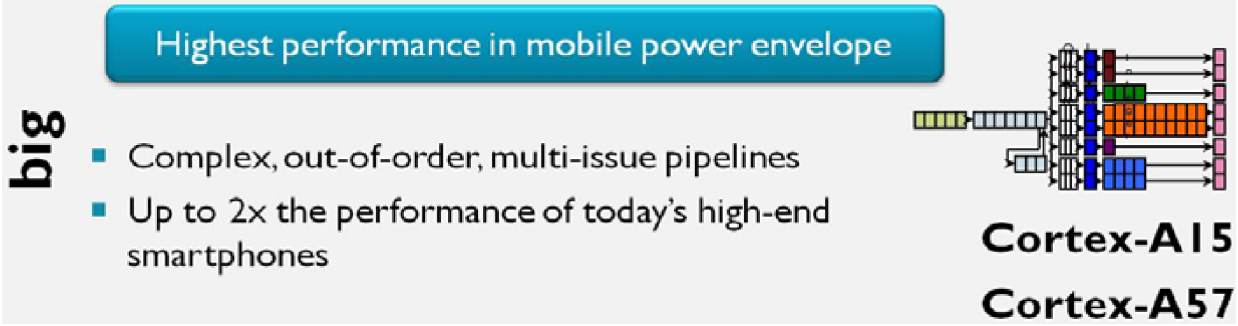
```
if (a==4 || a==10) x=0;
    CMP       r0,#4
    CMPNE     r0,#10
    MOVEQ     r1,#0
```

Big.LITTLE architecture

- ARM: Same architecture, different micro-architecture
  - Cortex A7:



  - Cortex A15:

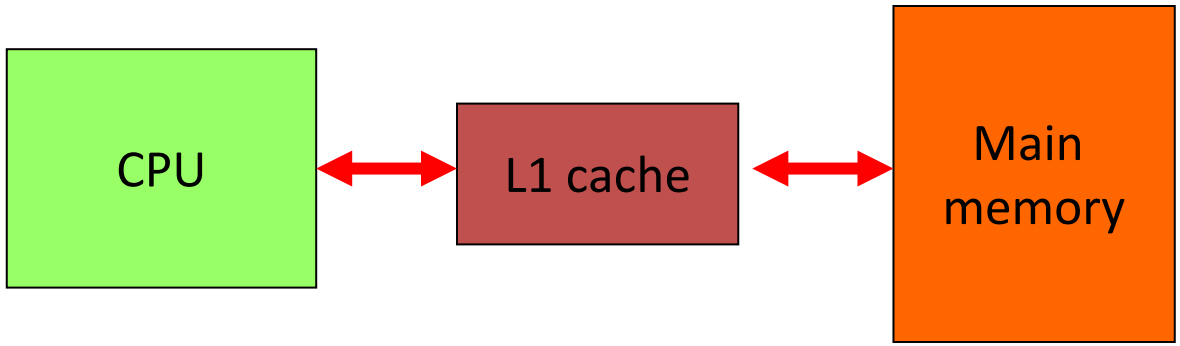


Memory System and Caches

- Memory is slower than CPU
  - CPU clock rates increase faster than memory
- Caches are used to speed up memory
  - Cache is a small but fast memory that holds copies of the contents of main memory
  - More expensive than main memory, but faster
- Memory Management Units (MMU)
  - Memory size is not large enough for all application?
  - Provide a larger virtual memory than physical memory

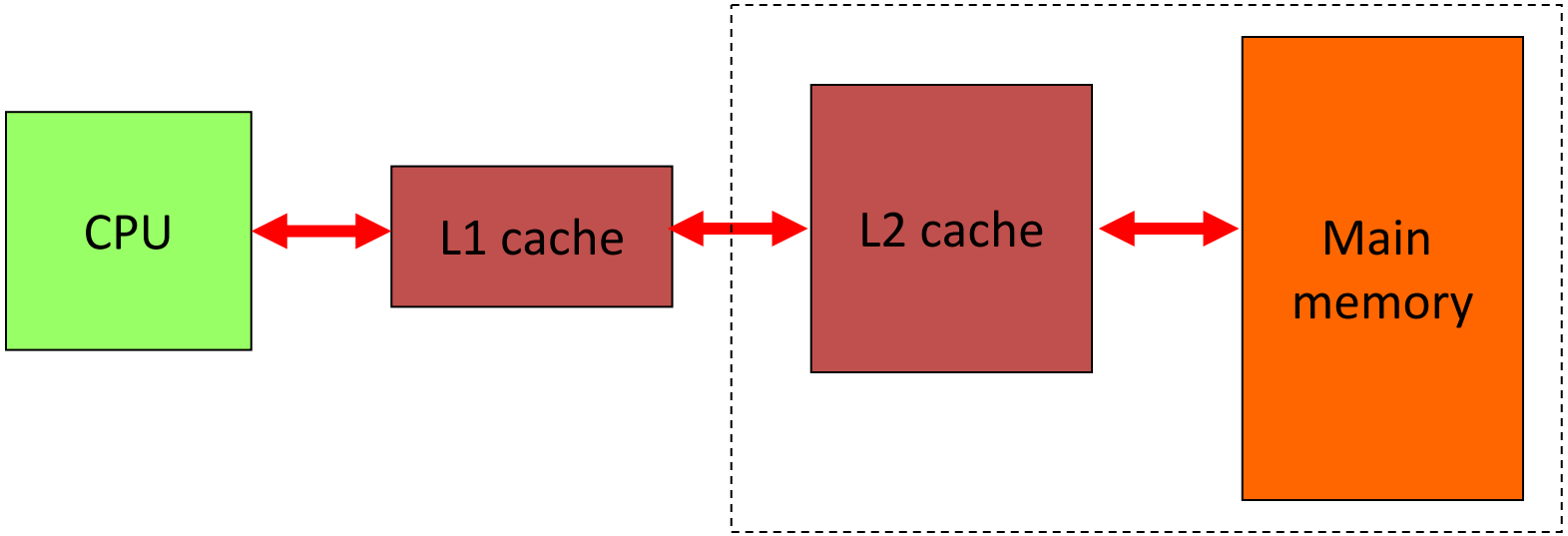
Memory System Performance

- h = cache hit rate: the percentage of cache hits
- t_{cache} = cache access time,
- t_{main} = main memory access time.
- Average memory access time:
  - $t_{av} = ht_{cache} + (1-h)t_{main}$
- Example: t_{cache} = 10ns, t_{main} = 100ns, h = 97%
  - t_{av} = 97%*10ns + (1-97%)*100ns = 12.7ns



Multi-Level Cache Access Time

- h_1 = cache hit rate for L1
- h_2 = cache hit rate for L2
- Average memory access time:
  - $t_{av} = h_1 t_{L1} + (1-h_1)(h_2 t_{L2} + (1-h_2)t_{main})$



Cache Organizations

- How should we map memory to cache?
  - Fully-associative: any memory location can be stored anywhere in the cache.
    - Ideal, best cache hit rate but implementation is complex and slow
    - Almost never implemented
  - Direct-mapped: each memory location maps onto exactly one cache entry.
    - Simplest, fastest but least flexible
    - Easy to have conflicts
  - N-way set-associative: each memory location can go into one of n sets.
    - Compromised solution

Memory Devices

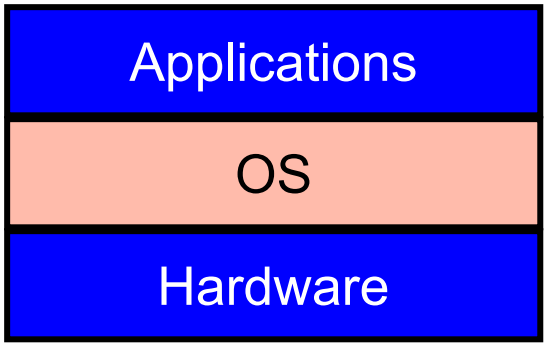
- Types of memory devices
  - RAM (Random-Access Memory)
    - Address can be read in any order, unlike magnetic disk/tape
    - Usually used for data storage
    - DRAM vs. SRAM.
  - ROM (Read-Only Memory)
    - Usually used or program storage
    - Mask-programmed vs. field-programmable.

RAM (Random-Access Memory)

- SRAM (Static RAM)
  - Faster, usually used for caches
  - Easier to integrate with logic.
  - Higher power consumption.
- DRAM (Dynamic RAM)
  - Structurally simpler
    - Only 1 transistor and 1 capacitor are required per bit, compared with 6 transistors used in SRAM
  - Can reach very high density
  - Must be periodically refreshed

What is an Operating System?

- Operating System
  - A software layer to abstract away and manage details of hardware resources
  - A set of utilities to simplify application development

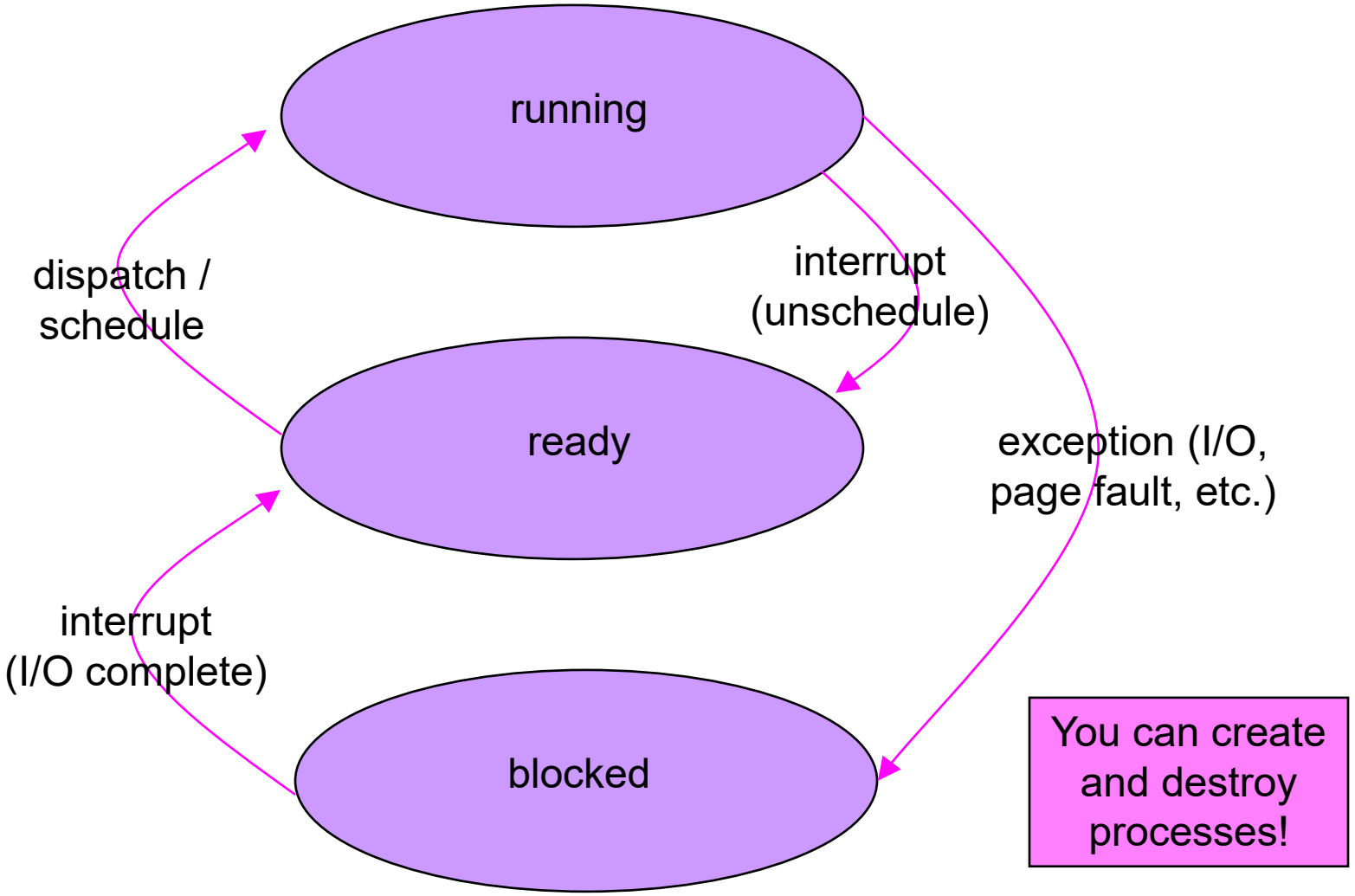


  - “all the code you didn’t write” in order to implement your application

Key OS Issues

- Multi-Programming
  - keeps multiple runnable jobs loaded in memory at once
  - overlaps I/O of a job with computing of another
    - while one job waits for I/O completion, OS runs instructions from another job
  - goal: optimize system throughput
    - perhaps at the cost of response time…
- Timesharing
  - multiple terminals into one machine
  - each user has illusion of entire machine to him/herself
  - optimize response time, perhaps at the cost of throughput
  - Timeslicing
    - divide CPU equally among the users

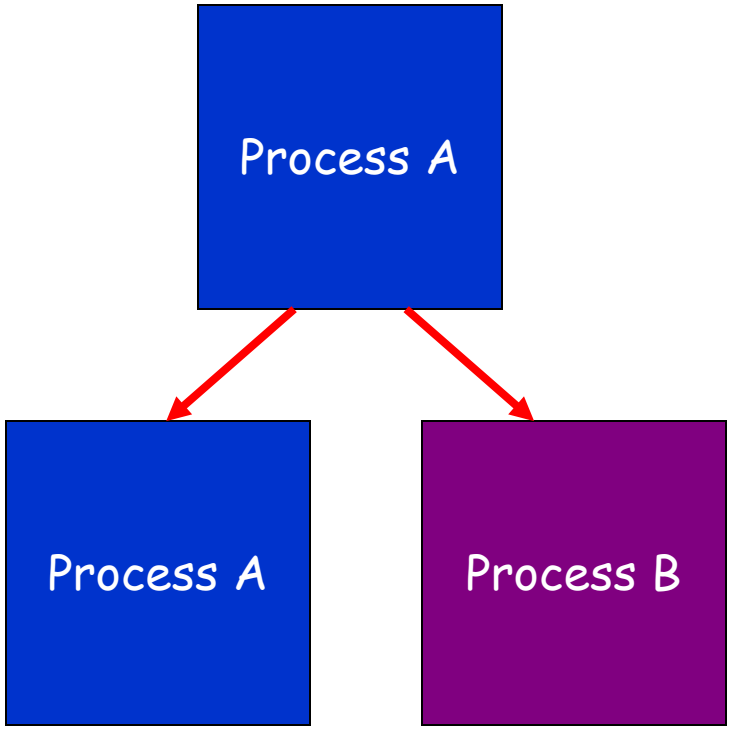
A Process's Lifecycle



Process Creation

Create a process with fork:

- parent process keeps executing the old program;
- child process executes a new program.



Create a New Process using fork()

- A process can create a child process by making a copy of itself
- Parent process is returned with the child process ID
- Child process gets a return value of 0

```
child_id = fork();
if (child_id == 0) {
  /* child operations */
} else {
  /* parent operations */
}
```

Overlay a Process using execv()

- Child process usually runs different code
- Use execv() to overlay the code of a process
- Parent uses wait() to wait for child process to finish and then release its memory

```
childid = fork();
if (childid == 0) {
   execv("mychild",childargs);
   exit(0);
} else {
   parent_stuff();
   wait(&csstatus);
   exit(0);
}
```

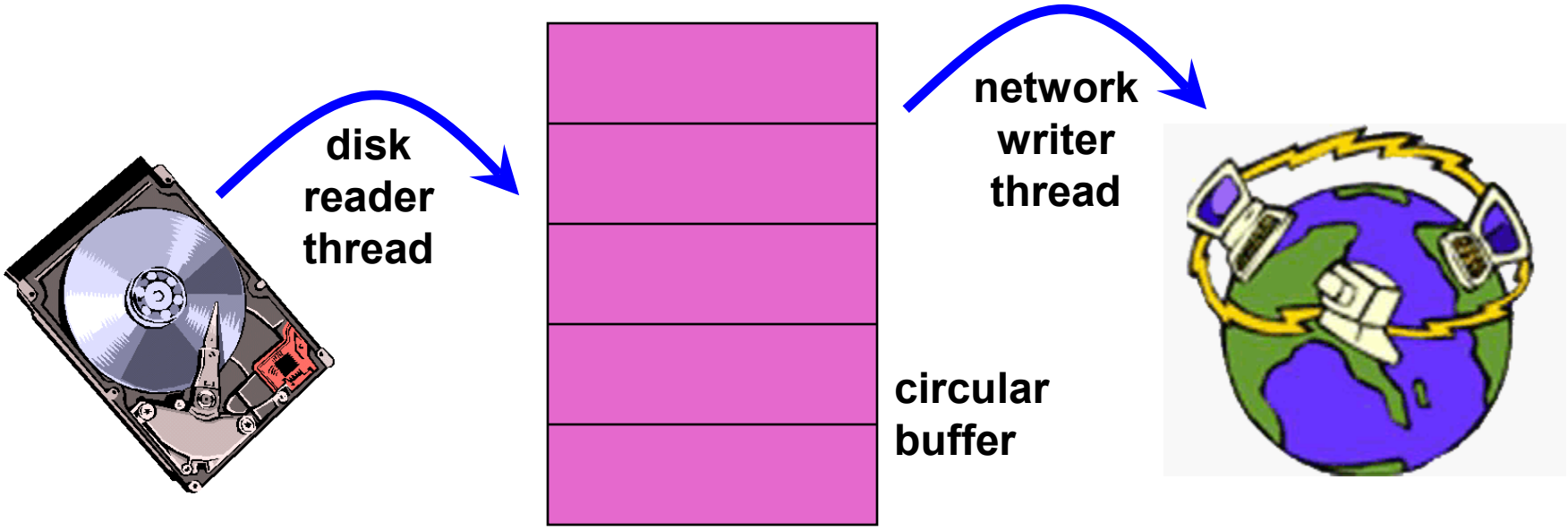
Thread

- A lightweight process
  - Separating the process's memory space
  - Better concurrency!

- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
  - creating concurrency does not require creating new processes
  - "faster / better / cheaper"

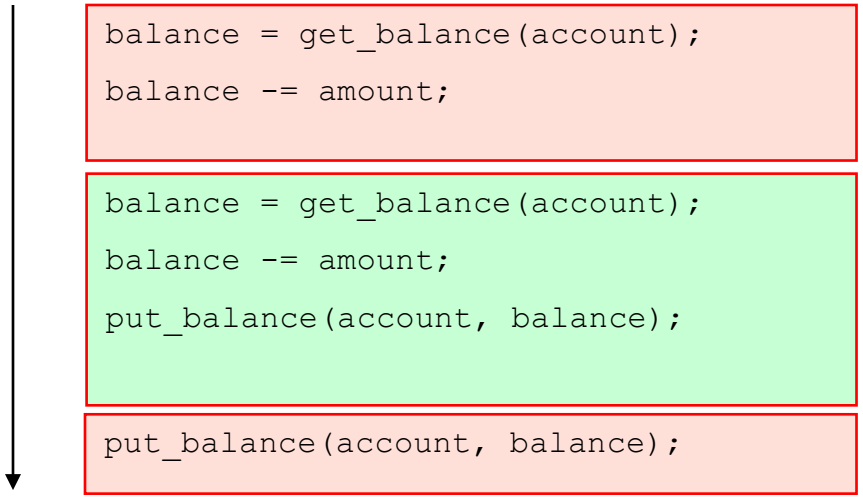
Synchronization

- Threads cooperate in multithreaded programs
  - to share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer



Interleaved Schedule

Execution sequence as seen by CPU



- Who comes first??

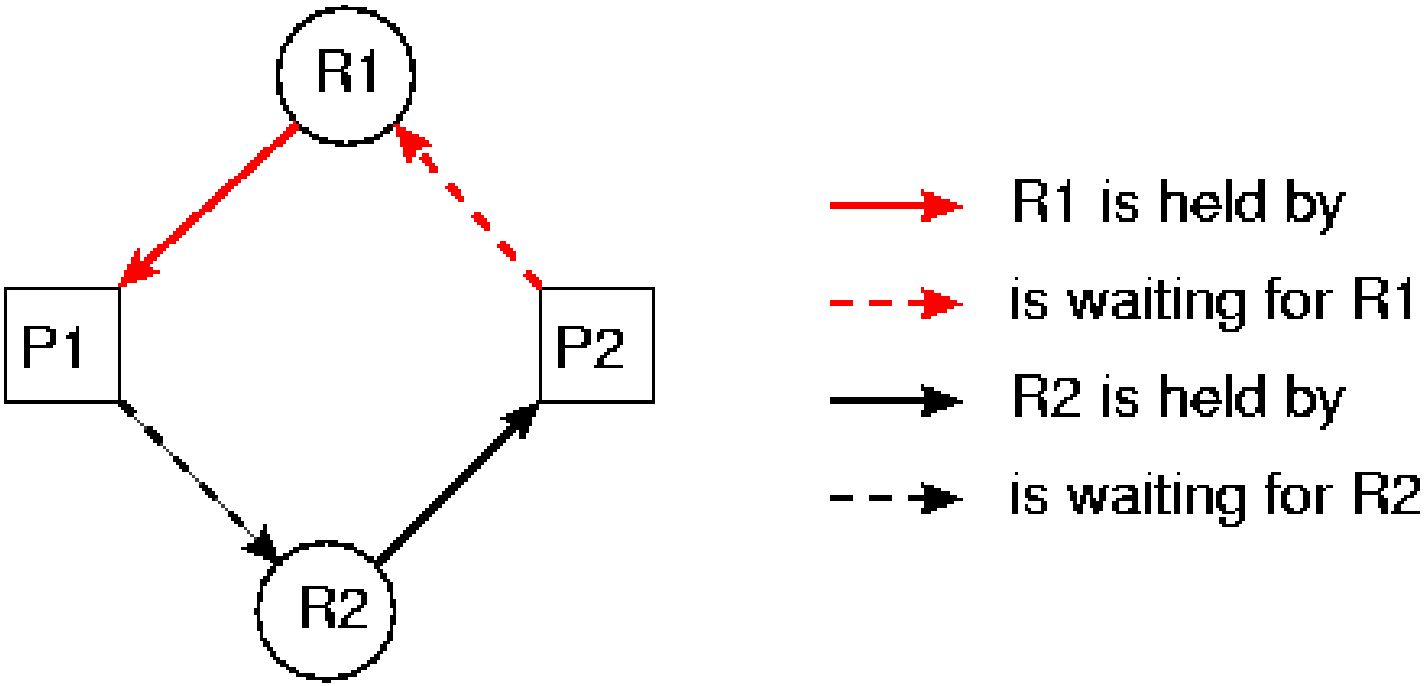
```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Deadlock

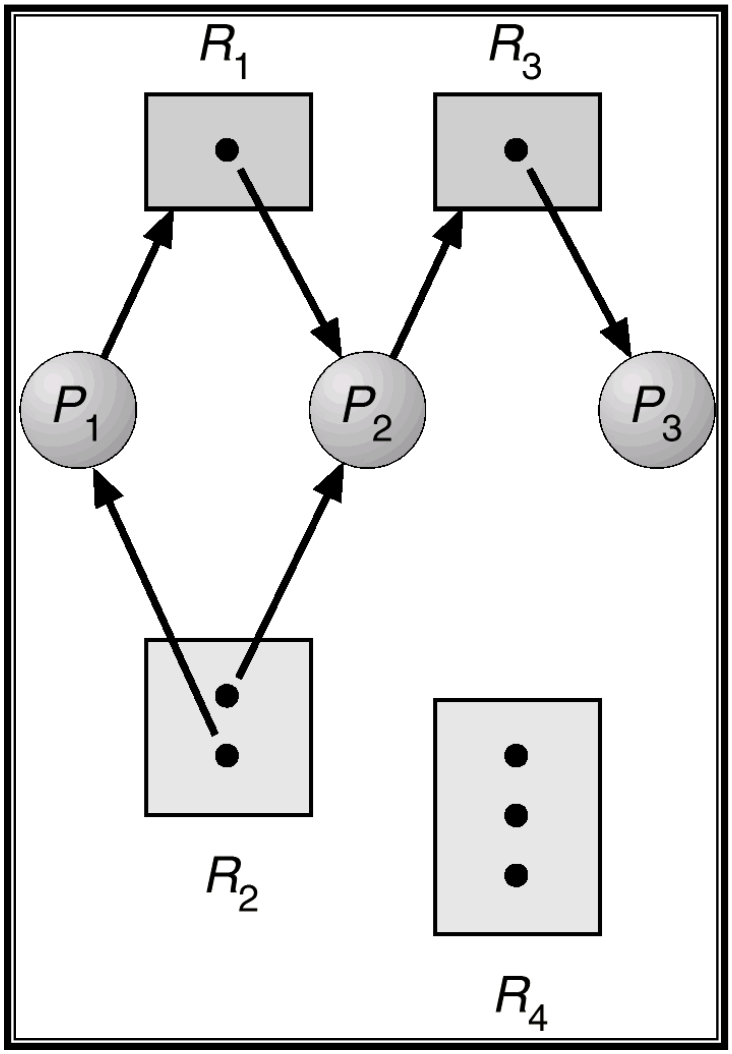
Deadlock

- Resource graph
  - A deadlock exists if there is an *irreducible cycle* in the resource graph



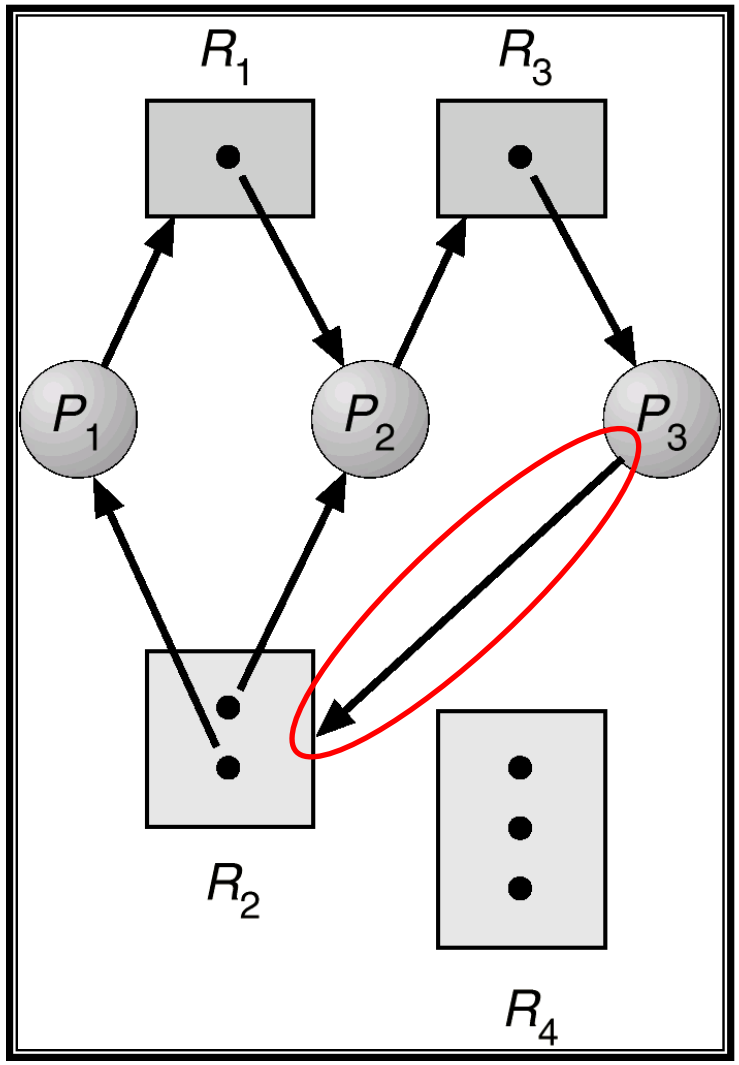
Resource Graph with No Cycle

- No deadlock



Resource Graph with A Deadlock

- An irreducible cycle



What is a reducible cycle?

- Resource graph with a cycle but no deadlock
- How to reduce?