بسم الله الرحمن الرحیم

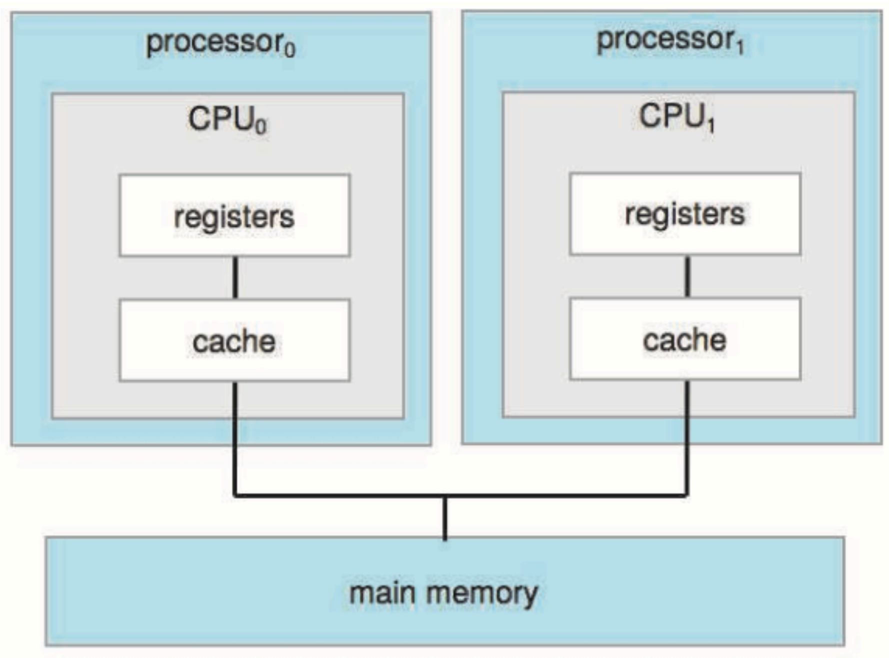
«سیستم عامل»

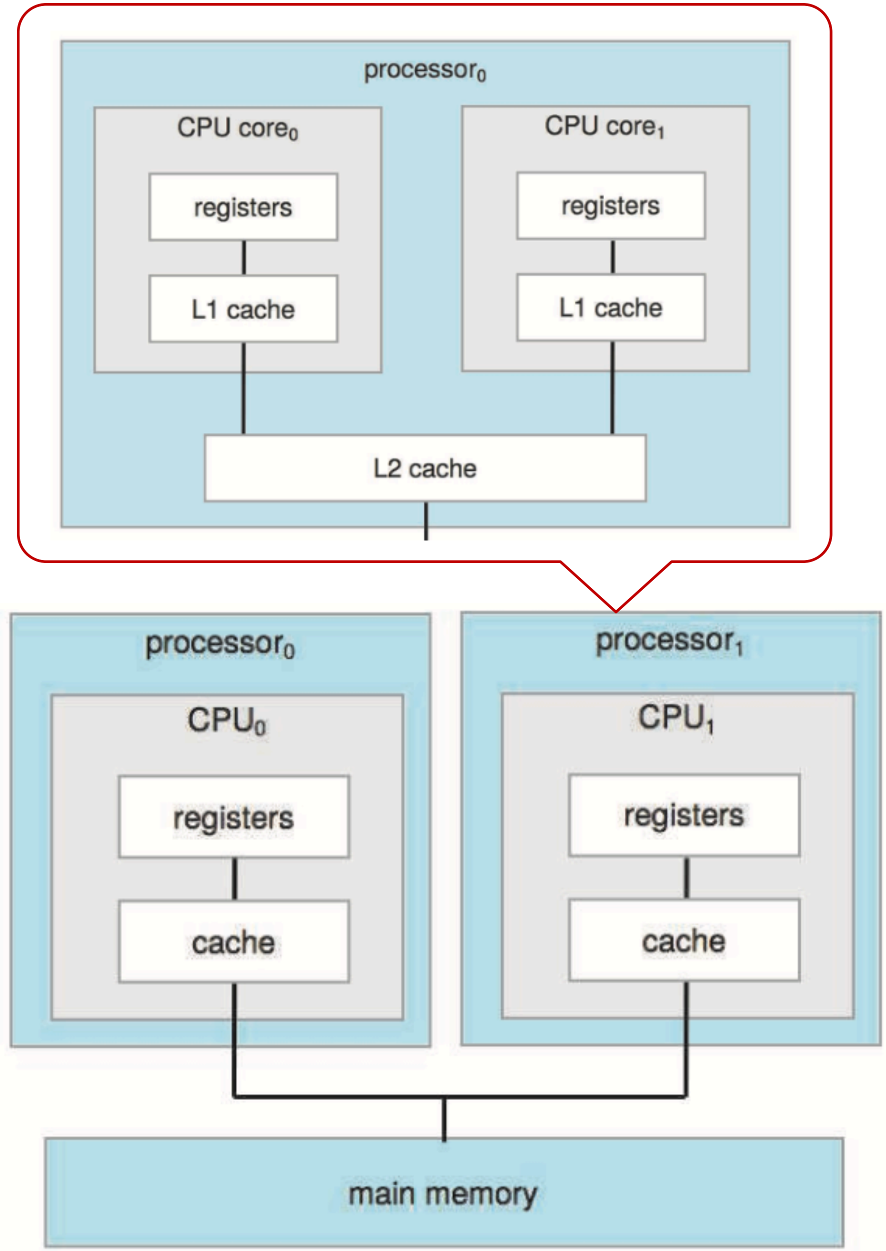
۱

جلسه ۲ : معرفی سیستم عامل: مروری بر سخت‌افزارهای مرتبط با سیستم عامل

مرور مطالب

- سخت‌افزارهای مرتبط
  - ۱) پردازنده
  - ۲) حافظه




processor$_0$
CPU core$_0$
CPU core$_1$
registers
registers
L1 cache
L1 cache
L2 cache
processor$_0$
processor$_1$
CPU$_0$
CPU$_1$
registers
registers
cache
cache
main memory

پردازنده

Instruction sets

- **A CPU's instruction set defines what it can do**
  - different for different CPU architectures
  - all have load and store instructions for moving items between memory and registers
    - Load a word located at an address in memory into a register
    - Store the contents of a register to a word located at an address in memory
  - many instructions for comparing and combining values in registers and putting result into a register
- **Look at the Blitz instruction set which is similar to a SUN SPARC instruction set**

Basic anatomy on a CPU

- Program Counter (PC)

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - Holds the instruction currently being executed

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**
  - hold variables and temporary results

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**
  - hold variables and temporary results
- **Arithmetic and Logic Unit (ALU)**

Basic anatomy on a CPU

- **Program Counter (PC)**
  - Holds the memory address of the next instruction
- **Instruction Register (IR)**
  - holds the instruction currently being executed
- **General Registers (Reg. 1..n)**
  - hold variables and temporary results
- **Arithmetic and Logic Unit (ALU)**
  - performs arithmetic functions and logic operations

Basic anatomy on a CPU

- **Stack Pointer (SP)**

Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables

Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables
- **Processor Status Word (PSW)**

Basic anatomy on a CPU

- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables
- **Processor Status Word (PSW)**
  - contains various control bits including the mode bit which determines whether privileged instructions can be executed at this time

Basic anatomy on a CPU

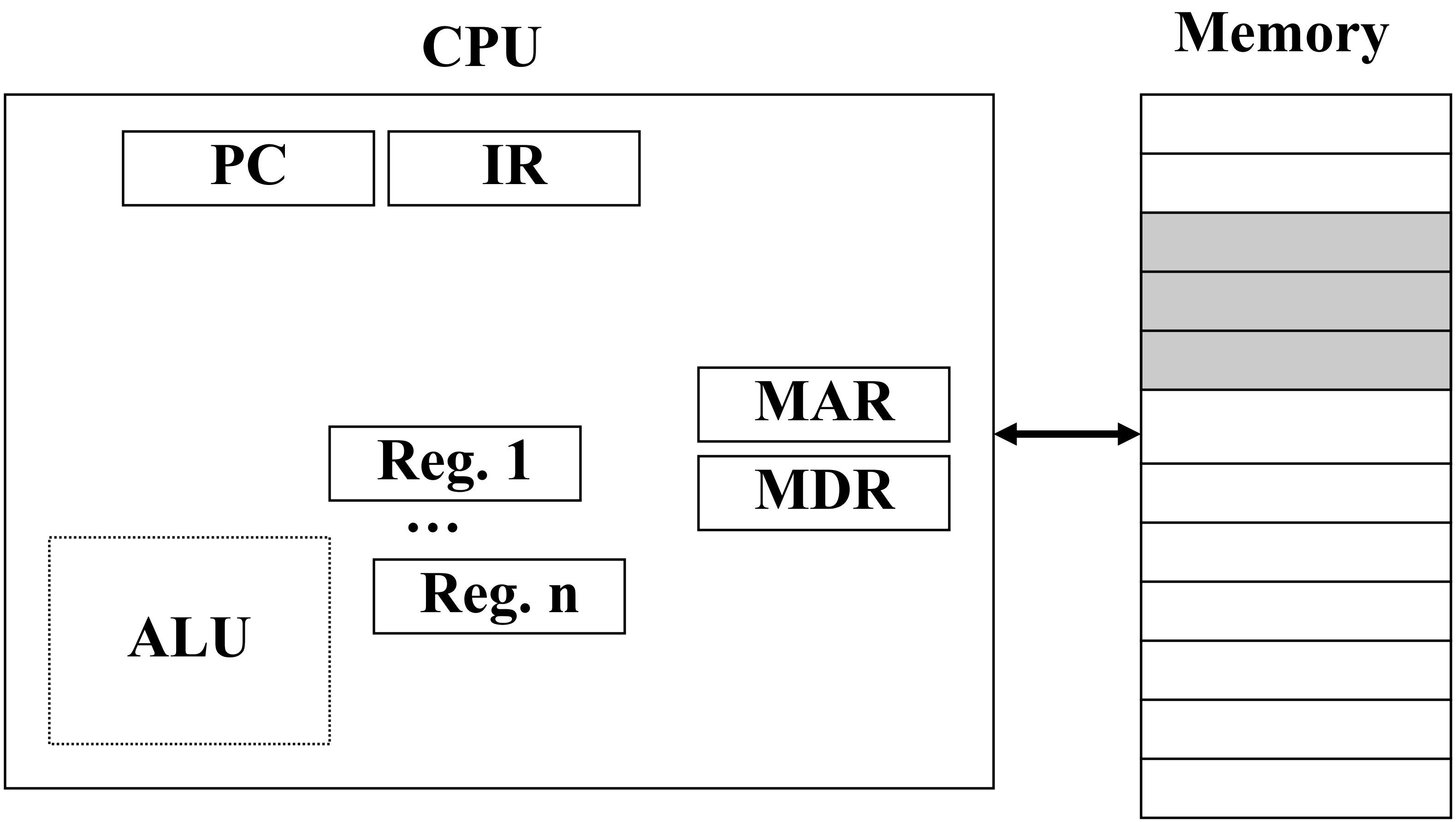
- **Stack Pointer (SP)**
  - holds memory address of a stack with a frame for each active procedure's parameters & local variables
- **Processor Status Word (PSW)**
  - contains various control bits including the mode bit which determines whether privileged instructions can be executed
- **Memory Address Register (MAR)**
  - contains address of memory to be loaded from/stored to
- **Memory Data Register (MDR)**
  - contains memory data loaded or to be stored

Program execution

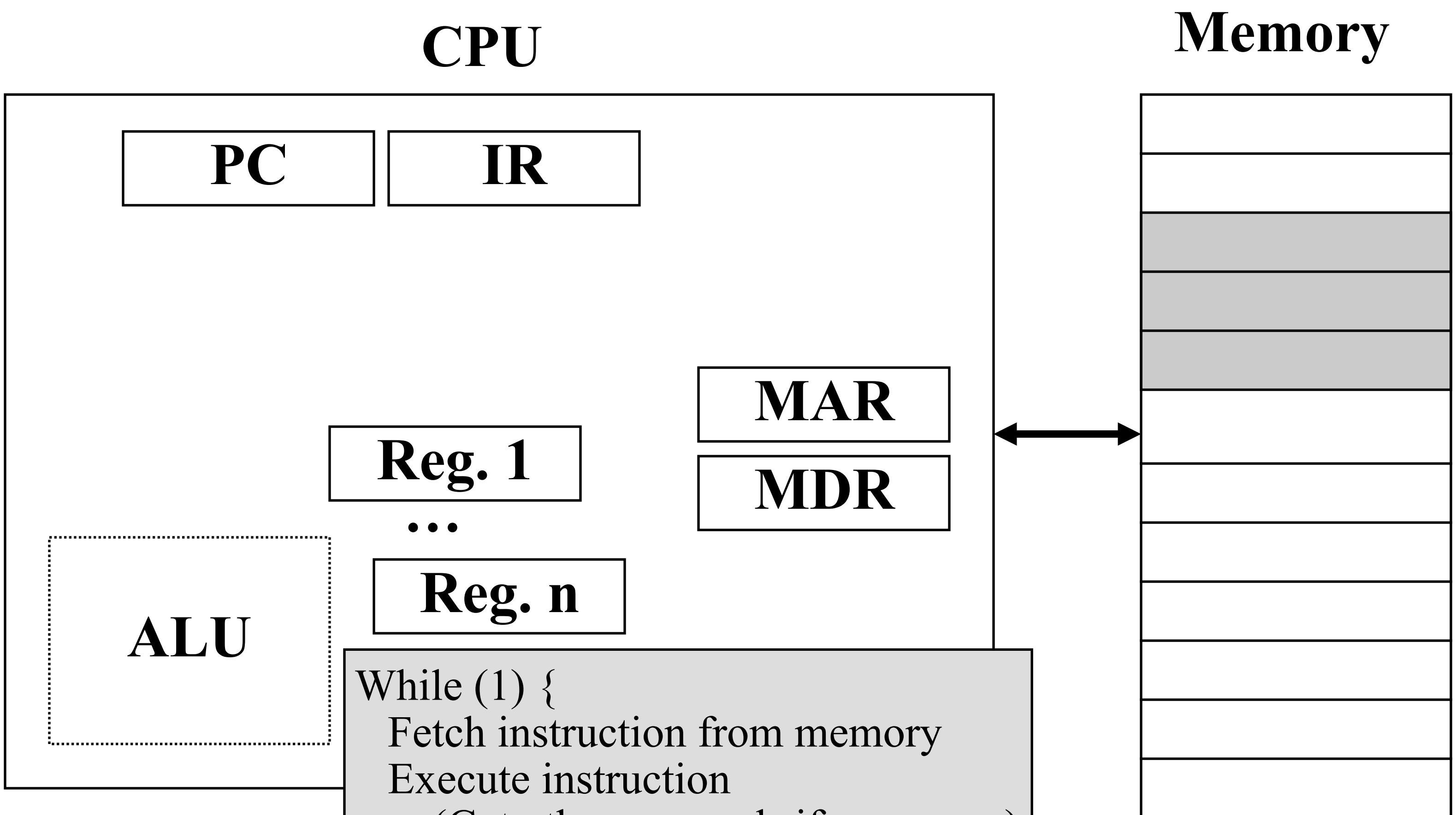
- **The Fetch/Decode/Execute cycle**
  - fetch next instruction pointed to by PC
  - decode it to find its type and operands
  - execute it
  - repeat

- **At a fundamental level, fetch/decode/execute is all a CPU does, regardless of which program it is executing**

Fetch/decode/execute cycle

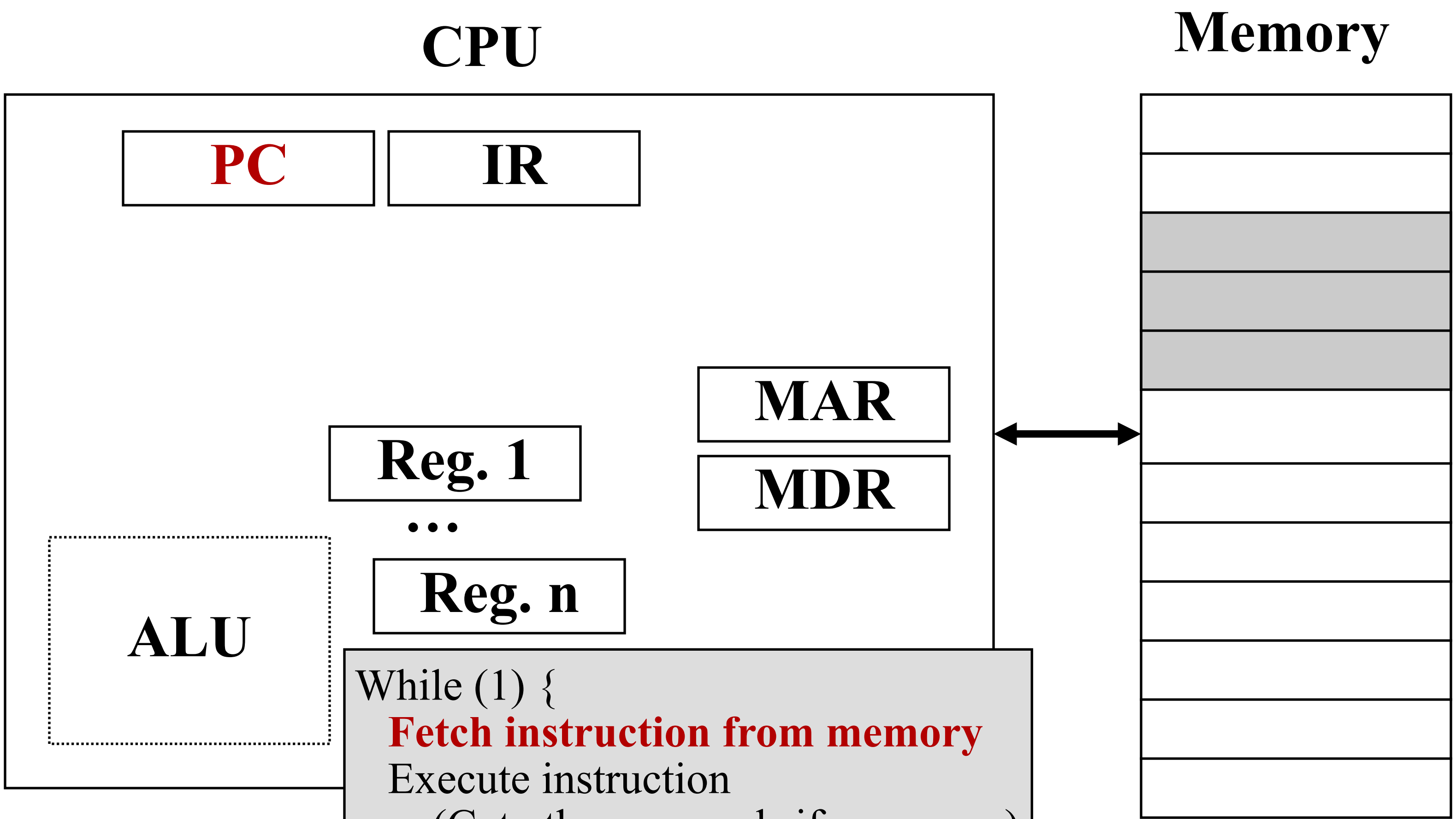


Fetch/decode/execute cycle



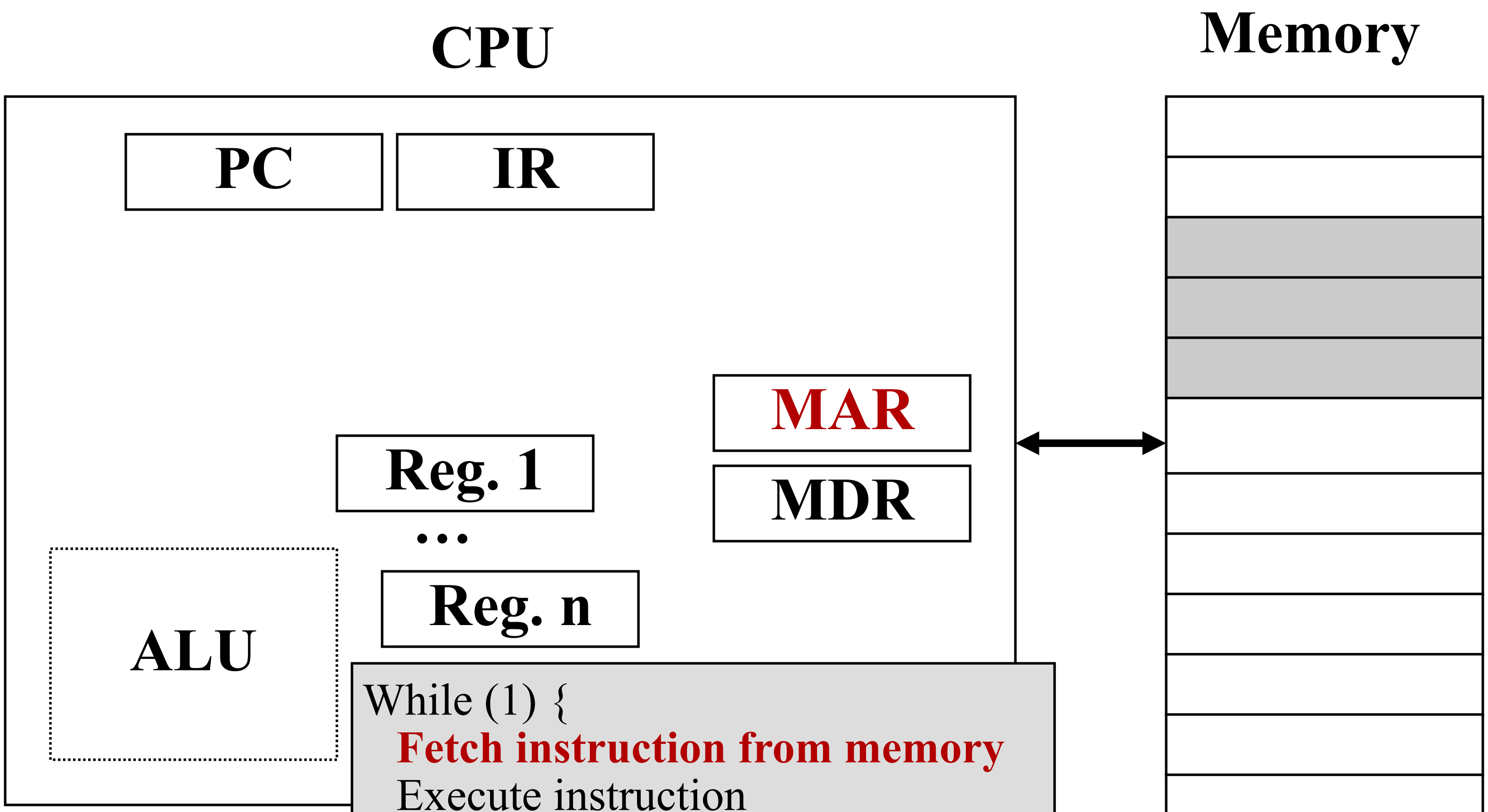

```
While (1) {
  Fetch instruction from memory
  Execute instruction
     (Get other operands if necessary)
  Store result
}
```

Fetch/decode/execute cycle



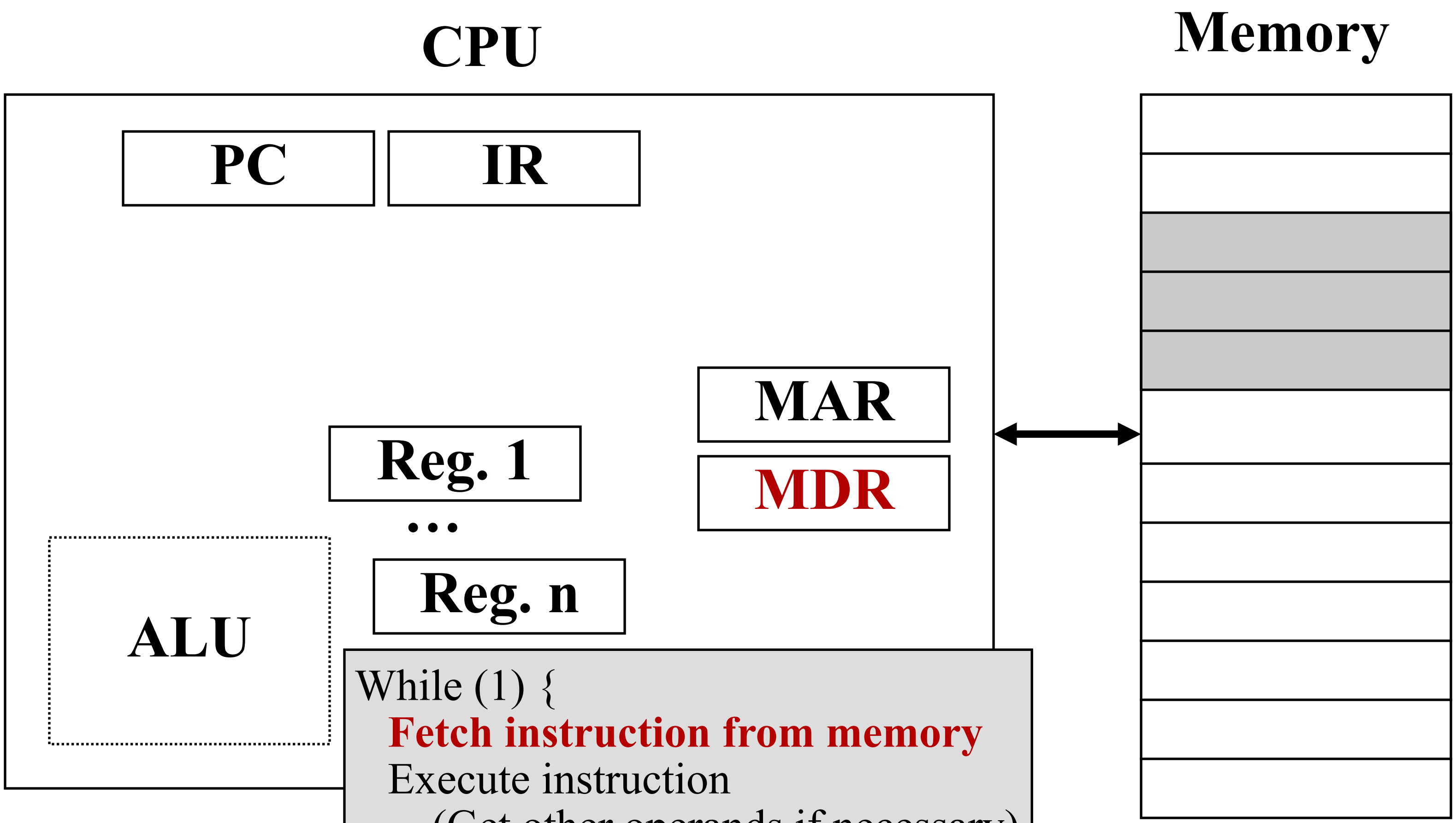

```
While (1) {
  Fetch instruction from memory
  Execute instruction
     (Get other operands if necessary)
  Store result
}
```

Fetch/decode/execute cycle



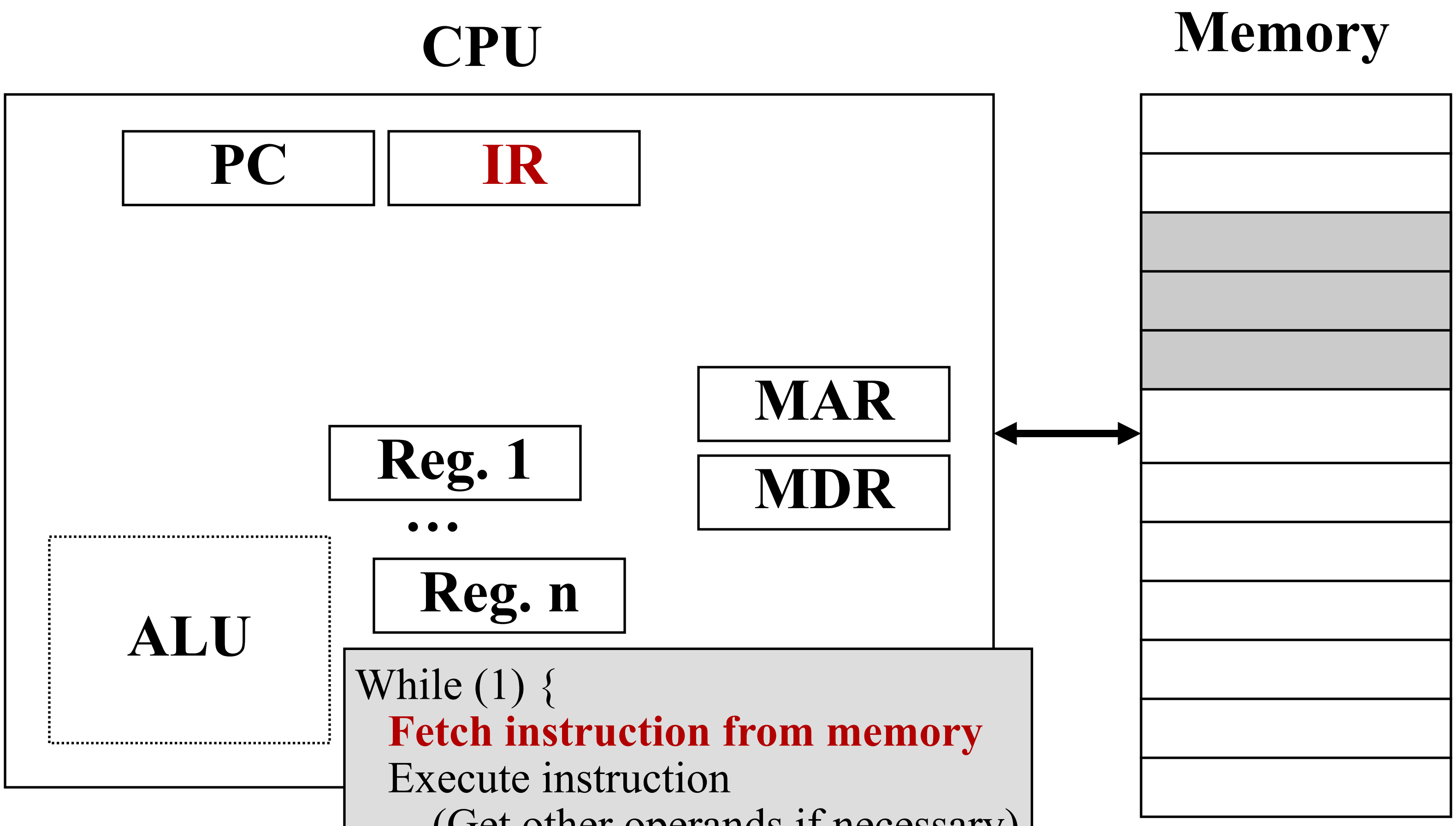
While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}

Fetch/decode/execute cycle



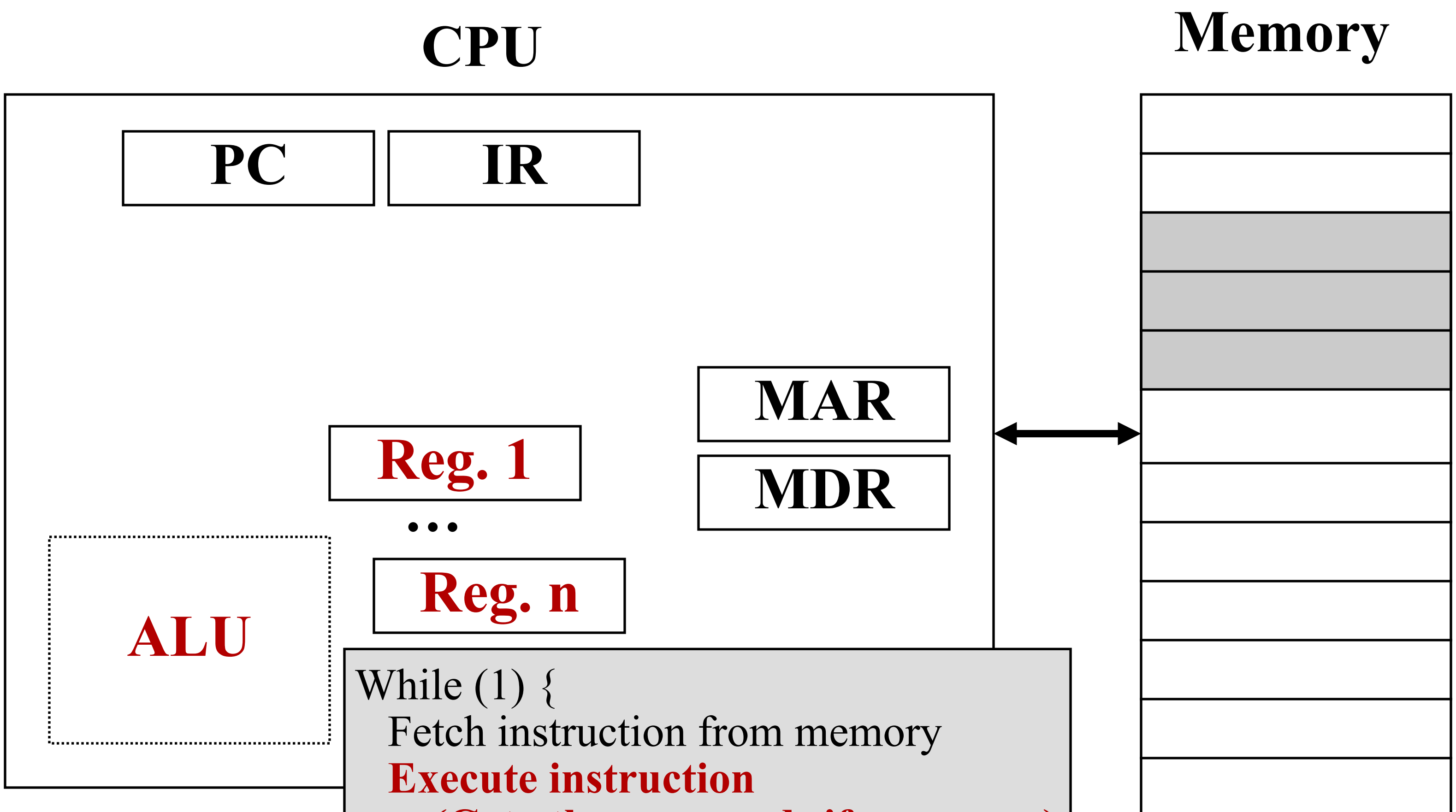
While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}

Fetch/decode/execute cycle



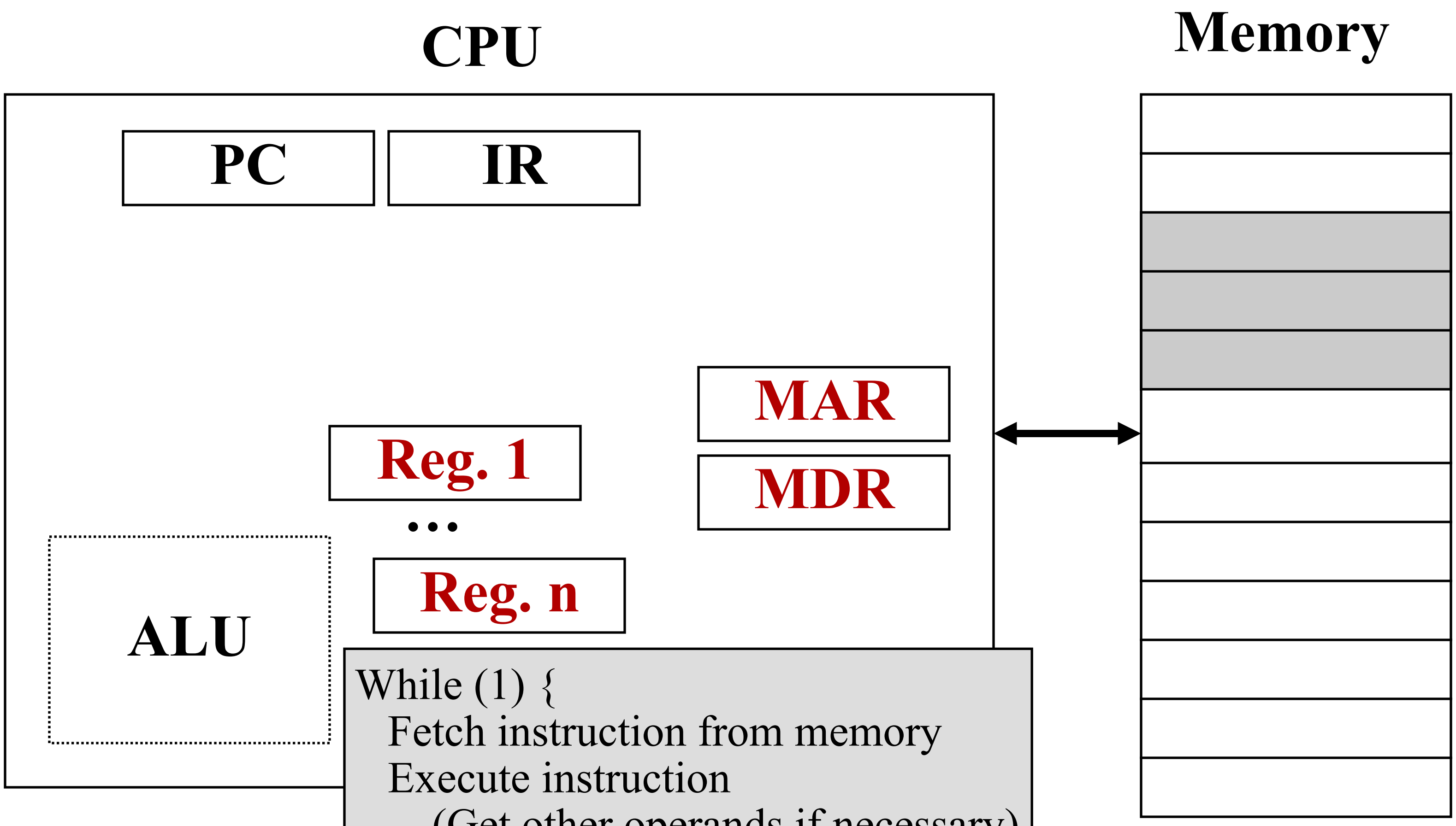

```
While (1) {
   Fetch instruction from memory
   Execute instruction
      (Get other operands if necessary)
   Store result
}
```

Fetch/decode/execute cycle



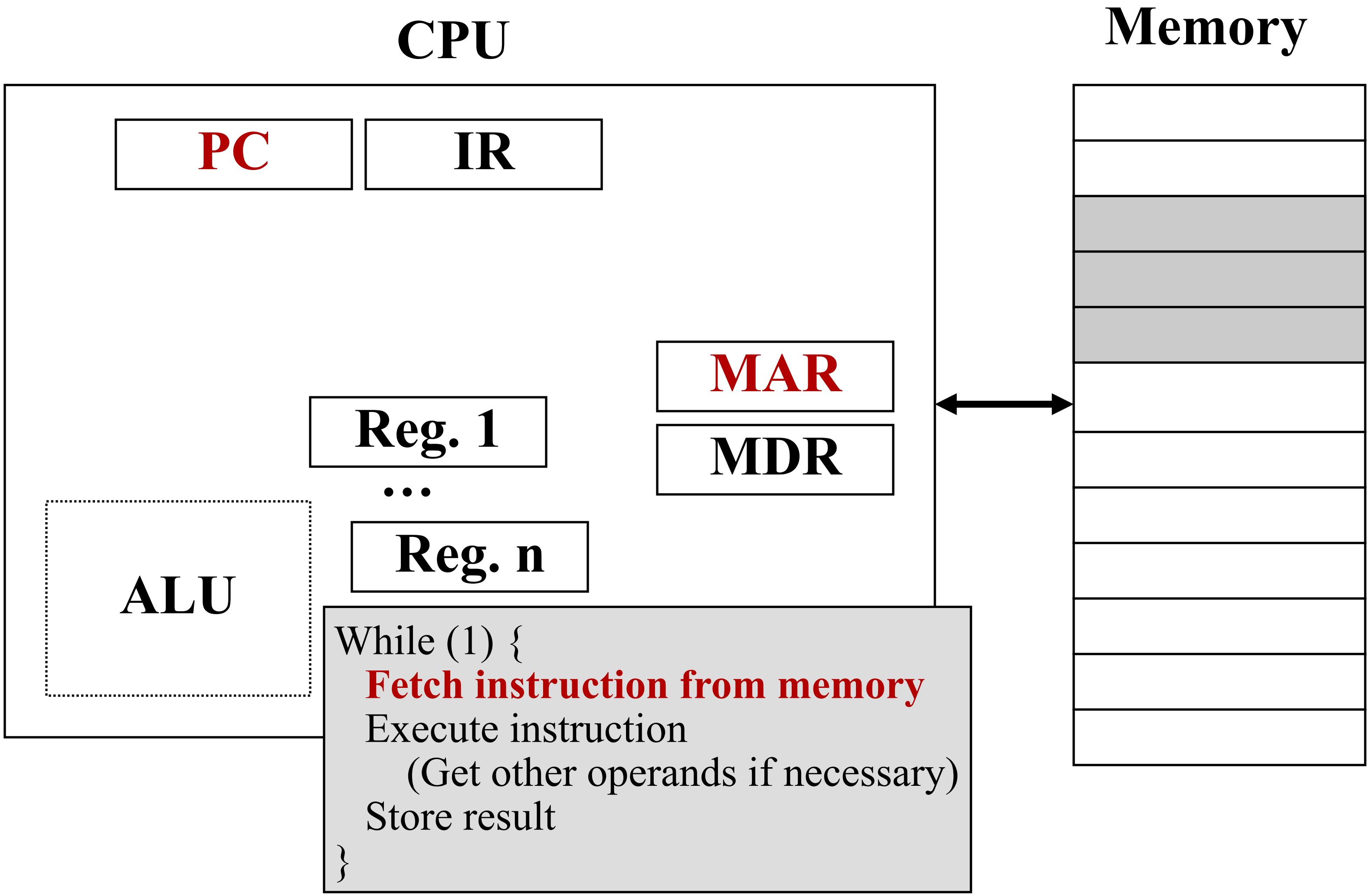
While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}

Fetch/decode/execute cycle




```
While (1) {
  Fetch instruction from memory
  Execute instruction
     (Get other operands if necessary)
  Store result
}
```

Fetch/decode/execute cycle



The OS is just a program!

The OS is just a program!

- **The OS is a sequence of instructions that the CPU will fetch/decode/execute**
  - How can the OS cause application programs to run?
  - How can the OS switch the CPU to run a different application and later resume the first one?
  - How can the OS maintain control?
  - In what ways can application code try to seize control indefinitely (ie. cheat)?
  - And how can the OS prevent such cheating?
  - How can applications programs cause the OS to run?

How can the OS invoke an application?

How can the OS invoke an application?

How can the OS invoke an application?

How can the OS invoke an application?

- **Somehow, the OS must load the address of the application's starting instruction into the PC**
  - The computer boots and begins running the OS
    - **OS code must be loaded into memory somehow**
    - **fetch/decode/execute OS instructions**
    - **OS requests user input to identify application "file"**
    - **OS loads application file (executable) into memory**
    - **OS loads the memory address of the application's starting instruction into the PC**
    - **CPU fetches/decodes/executes the application's instructions**

How can OS guarantee to regain control?

How can OS guarantee to regain control?

- What if a running application doesn't make a system call and hence hogs the CPU?

How can OS guarantee to regain control?

- **What if a running application doesn't make a system call and hence hogs the CPU?**
  - OS needs interrupts from a timer device!

How can OS guarantee to regain control?

- **What if a running application doesn't make a system call and hence hogs the CPU?**
  - OS needs interrupts from a timer device!
  - OS must register a future timer interrupt before it hands control of the CPU over to an application

How can OS guarantee to regain control?

- **What if a running application doesn't make a system call and hence hogs the CPU?**
  - OS needs interrupts from a timer device!
  - OS must register a future timer interrupt before it hands control of the CPU over to an application
  - When the timer interrupt goes off the interrupt hardware jumps control back into the OS at a pre-specified location called an interrupt handler

How can OS guarantee to regain control?

- **What if a running application doesn't make a system call and hence hogs the CPU?**
  - OS needs interrupts from a timer device!
  - OS must register a future timer interrupt before it hands control of the CPU over to an application
  - When the timer interrupt goes off the interrupt hardware jumps control back into the OS at a pre-specified location called an interrupt handler
  - The interrupt handler is just a program (part of the OS)

How can OS guarantee to regain control?

- **What if a running application doesn't make a system call and hence hogs the CPU?**
  - OS needs interrupts from a timer device!
  - OS must register a future timer interrupt before it hands control of the CPU over to an application
  - When the timer interrupt goes off the interrupt hardware jumps control back into the OS at a pre-specified location called an interrupt handler
  - The interrupt handler is just a program (part of the OS)
  - The address of the interrupt handler's first instruction is placed in the PC by the interrupt h/w

What if the application tries to cheat?

What if the application tries to cheat?

- What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications
  - The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications
  - The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!
- Privileged **instructions can only be executed when the mode bit is set**

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications
  - The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!
- Privileged **instructions can only be executed when the mode bit is set**
  - disabling interrupts

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications
  - The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!
- Privileged **instructions can only be executed when the mode bit is set**
  - disabling interrupts
  - setting the mode bit!

What if the application tries to cheat?

- **What stops the running application from disabling the future timer interrupt so that the OS can not take control back from it?**
  - Disabling interrupts must be a privileged instruction which is not executable by applications
  - The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!
- Privileged **instructions can only be executed when the mode bit is set**
  - disabling interrupts
  - setting the mode bit!
  - Attempted execution in non-privileged mode generally causes an interrupt (trap) to occur

What other ways are there to cheat?

What other ways are there to cheat?

- What stops the running application from modifying the OS?

What other ways are there to cheat?

- **What stops the running application from modifying the OS?**
  - Specifically, what stops it from modifying the timer interrupt handler to jump control back to the application?

What other ways are there to cheat?

- **What stops the running application from modifying the OS?**
  - Memory protection!
  - Memory protection instructions must be privileged
  - They can only be executed with the mode bit set …
- **Why must the OS clear the mode bit before it hands control to an application?**

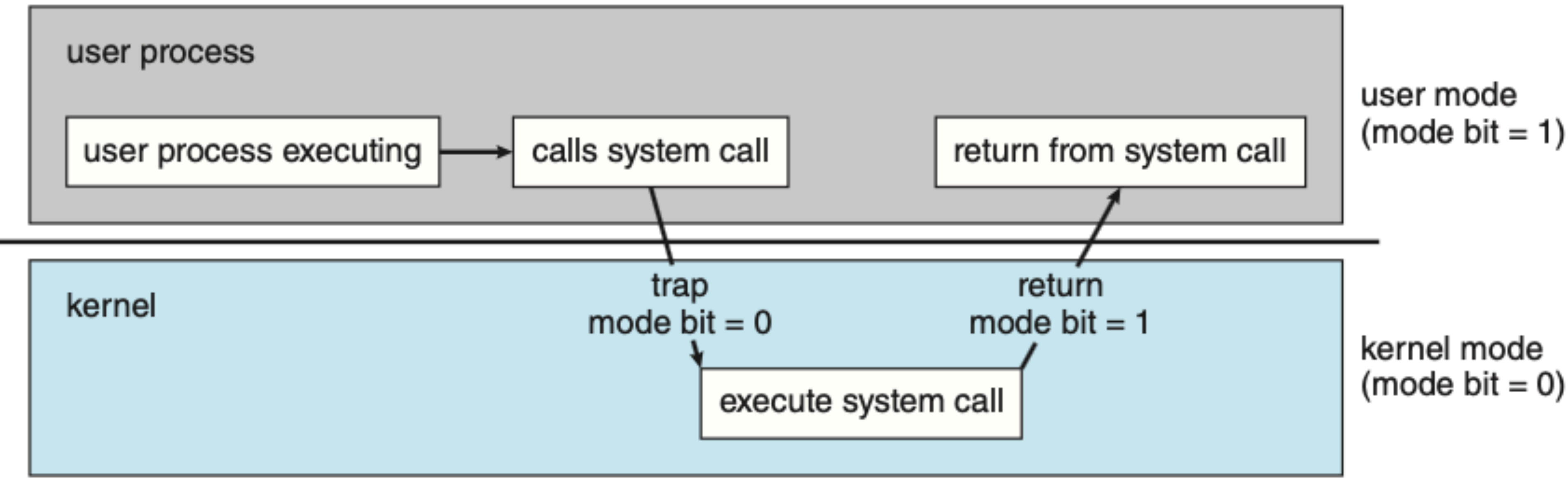
How can applications invoke the OS?

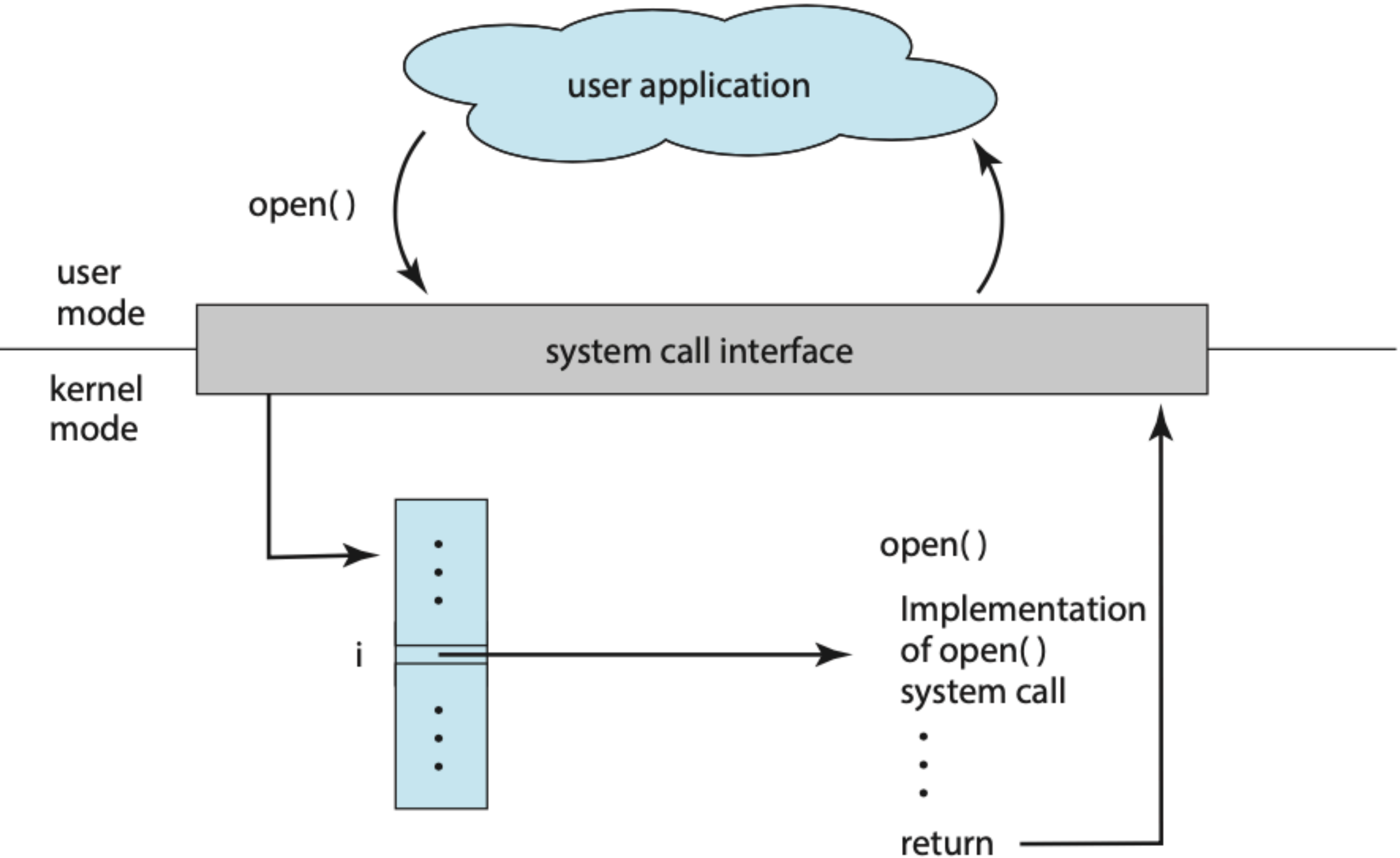
How can applications invoke the OS?

- Why not just set PC to an OS instruction address and transfer control that way?

How can applications invoke the OS?

- **Special instruction causes a trap / interrupt**
- **Trap instruction changes PC to point to a predetermined OS entry point instruction and simultaneously sets the mode bit**
  - application calls a library procedure that includes the appropriate trap instruction
  - fetch/decode/execute cycle begins at a pre-specified OS entry point called a system call
  - CPU is now running in privileged mode
- **Traps, like interrupts, are hardware events, but they are caused by the executing program rather than a device external to the CPU**


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)


user application
open( )
user
mode
system call interface
kernel
mode
i
open( )
Implementation
of open( )
system call
return

How can the OS switch to a new application?

How can the OS switch to a new application?

- **To suspend execution of an application simply capture its memory state and processor state**
  - preserve all the memory values of this application
  - copy values of all CPU registers into a data structure which is saved in memory
  - restarting the application from the same point just requires reloading the register values

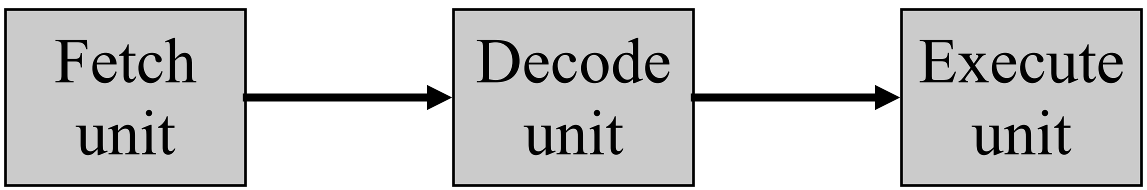
Recap

- Why do we need a timer device?
- Why do we need an interrupt mechanism?
- Why are system calls different to procedure calls?
- How are system calls different to interrupts?
- Why is memory protection necessary?

Why its not quite that simple ...

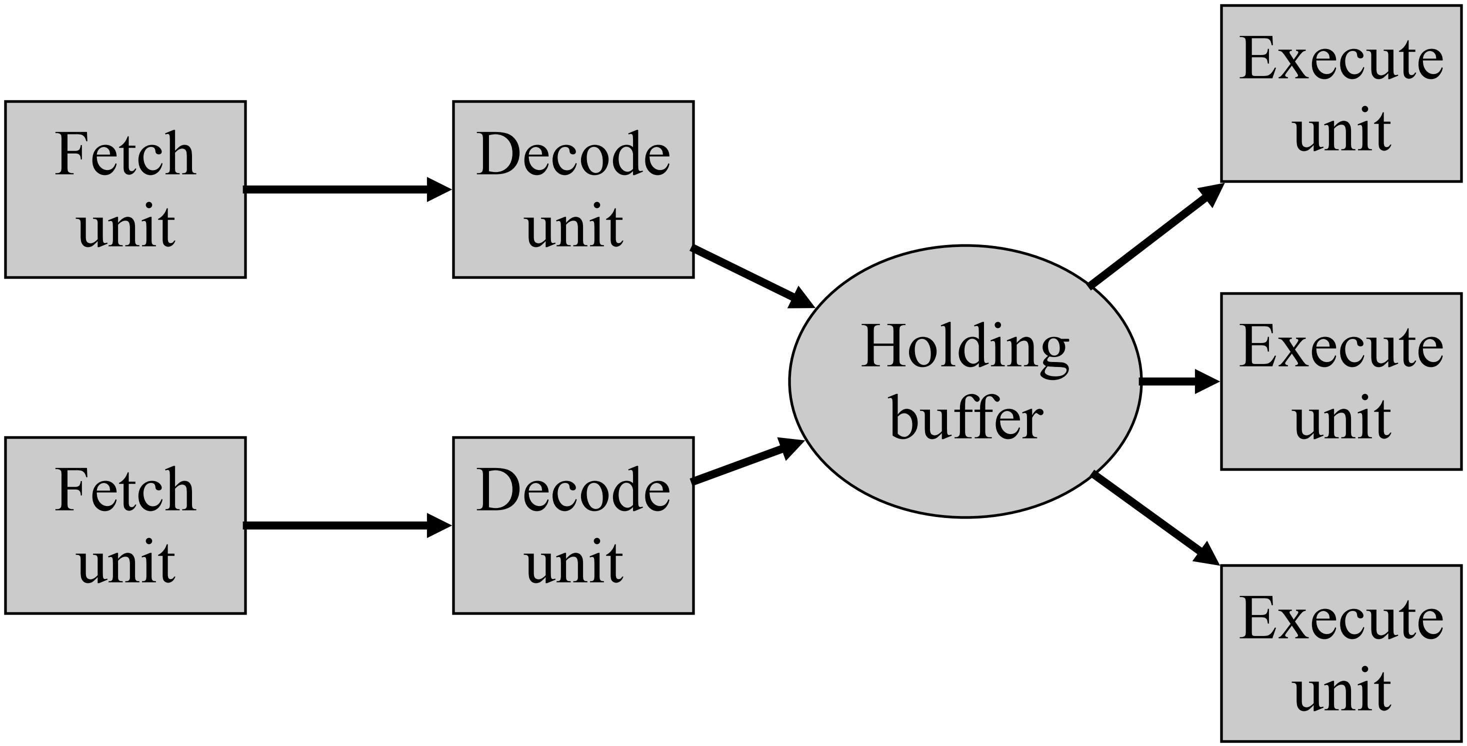
- Pipelined CPUs
- Superscalar CPUs
- Multi-level memory hierarchies
- Virtual memory
- Complexity of devices and buses
- Heterogeneity of hardware

Pipelined CPUs



Execution of current instruction performed in parallel with decode of next instruction and fetch of the one after that

Superscalar CPUs



What does this mean for the OS?

What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in capturing state of a running application
  - more expensive to suspend and resume applications

What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in capturing state of a running application
  - more expensive to suspend and resume applications
- **Superscalar CPUs**
  - even more complexity in capturing state of a running application
  - even more expensive to suspend and resume applications

What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in capturing state of a running application
  - more expensive to suspend and resume applications
- **Superscalar CPUs**
  - even more complexity in capturing state of a running application
  - even more expensive to suspend and resume applications

- **More details, but fundamentally the same task**

What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in capturing state of a running application
  - more expensive to suspend and resume applications
- **Superscalar CPUs**
  - even more complexity in capturing state of a running application
  - even more expensive to suspend and resume applications

- **More details, but fundamentally the same task**
- **The BLITZ CPU is not pipelined or superscalar**

حافظه

The memory hierarchy

- 2GHz processor → 0.5 ns
- Data/inst. cache → 0.5ns – 10 ns, 64 kB- 1MB (this is where the CPU looks first!)
- Main memory → 60 ns, 512 MB – 1GB
- Magnetic disk → 10 ms, 160 Gbytes

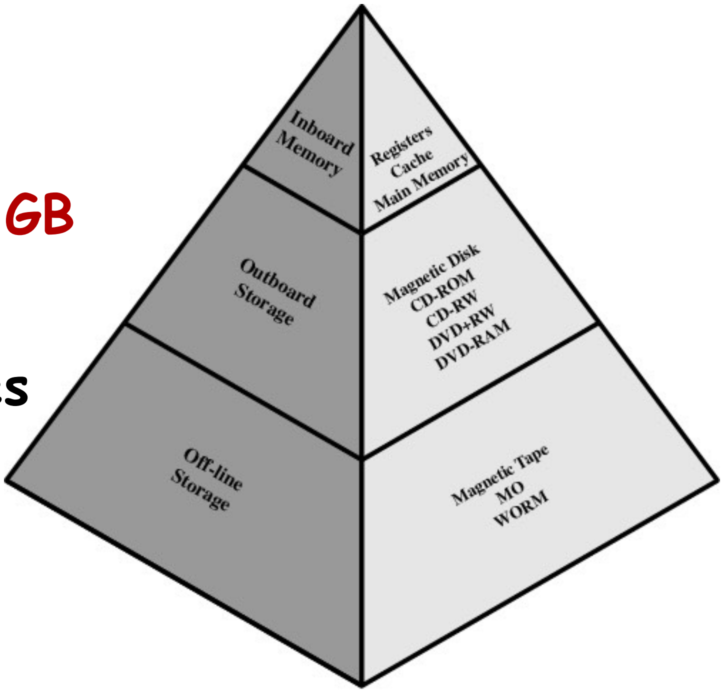


Figure 1.14 The Memory Hierarchy

Who manages the memory hierarchy?

Who manages the memory hierarchy?

- **Movement of data from main memory to cache is under hardware control**
  - cache lines loaded on demand automatically
  - replacement policy fixed by hardware

Who manages the memory hierarchy?

- **Movement of data from main memory to cache is under hardware control**
  - cache lines loaded on demand automatically
  - replacement policy fixed by hardware
- **Movement of data from cache to main memory can be affected by OS**
  - instructions for “flushing” the cache
  - can be used to maintain consistency of main memory

Who manages the memory hierarchy?

- **Movement of data from main memory to cache is under hardware control**
  - cache lines loaded on demand automatically
  - replacement policy fixed by hardware

- **Movement of data from cache to main memory can be affected by OS**
  - instructions for "flushing" the cache
  - can be used to maintain consistency of main memory

- **Movement of data among lower levels of the memory hierarchy is under direct control of the OS**
  - virtual memory page faults
  - file system calls

OS implications of a memory hierarchy?

OS implications of a memory hierarchy?

- How do you keep the contents of memory consistent across layers of the hierarchy?
- How do you allocate space at layers of the memory hierarchy "fairly" across different applications?

OS implications of a memory hierarchy?

- How do you keep the contents of memory consistent across layers of the hierarchy?
- How do you allocate space at layers of the memory hierarchy “fairly” across different applications?
- How do you hide the latency of the slower subsystems?
  - These include main memory as well as disk!
- How do you protect one application’s area of memory from other applications?

OS implications of a memory hierarchy?

- How do you keep the contents of memory consistent across layers of the hierarchy?
- How do you allocate space at layers of the memory hierarchy “fairly” across different applications?
- How do you hide the latency of the slower subsystems?
    - These include main memory as well as disk!
- How do you protect one application’s area of memory from other applications?
- How do you relocate an application in memory?

Quiz

How does the OS solve these problems:

- Time sharing the CPU among applications?
- Space sharing the memory among applications?
- Protection of applications from each other?
- Protection of hardware/devices?
- Protection of the OS itself?