Design and Engineering of Computer Systems

Lecture 3:
Overview of CPU hardware

Mythili Vutukuru
IIT Bombay
CPU hardware

- User program = code (instructions for CPU) + data
- Stored program concept
  - User programs stored in main memory (RAM)
  - CPU fetches code/data from RAM and executes instructions
- CPU runs processes = running programs
- Modern CPUs have multiple CPU cores for parallel execution
  - Each CPU core runs one process at a time each
  - Modern CPUs have hyperthreading, where each core can run more than one process also
Instructions and registers

- Every CPU has
  - A set of **instructions** that the hardware can execute
  - A set of **registers** for temporary storage of data
- Defined by **ISA** = Instruction Set Architecture
  - Specific to CPU manufacturer (e.g., Intel’s x86 ISA)
- **Registers**: special registers (specific purpose) or general purpose
  - **Program counter (PC)** is a special register, has memory address of the next instruction to execute on the CPU
  - General purpose registers can be used for anything, e.g., operands in instructions
- Size of registers defined by architecture (32 bit / 64 bit)
  - 32-bit PC can store addresses of $2^{32} = 4\text{GB}$ of memory
CPU instructions

• Some common examples of CPU instructions
  • Load: copy content from memory location → register
  • Store: copy content from register → memory location
  • Arithmetic operations like add: reg1 + reg2 → reg3
  • Logical operations, compare, ...
  • Jump: set PC to specific value

• Simple model of CPU
  • Each clock cycle, fetch instruction at PC, decode, access required data, execute, update PC, repeat
  • PC incremented to next instruction, or jump to specific value

• Many optimizations to this simple model
  • Pipelining: run multiple instructions concurrently in a pipeline
  • Many more in modern CPUs to optimize #instructions executed per clock cycle
Running a program

• What happens when you run a C program?
  • C code translated into **executable** = instructions that the CPU can understand
  • Translation done by program called **compiler**
  • Executable file stored on hard disk (say, “a.out”)
  • When executable is run, a new **process** is created in RAM
  • **Memory image** of process in RAM contains code+data (and other things)
  • CPU starts executing the instructions of the program

• When CPU core is running a process, CPU registers contain the execution **context** of the process
  • PC points to instruction in the program, general purpose registers store data in the program, and so on
Concurrent execution

- CPU runs multiple programs concurrently
  - Run one process, switch to another, switch again, ...
- How to ensure correct concurrent execution?
  - Run process P1 for some time
  - Pause P1, save context somewhere in memory (more later)
  - Load context of P2 from memory
  - Run P2 for some time
  - Pause P2, save context of P2, restore context of P1, switch to P1
- Every process thinks it is running alone on CPU
  - Saving and restoring context ensures process sees no disruption
- Operating System (OS) takes care of this switching across processes
  - OS virtualizes CPU across multiple processes
Interrupt handling

• In addition to running user programs, CPU also has to handle external events (e.g., mouse click, keyboard input)

• **Interrupt** = external signal from I/O device asking for CPU’s attention

• How are interrupts handled?
  • CPU is running process P1 and interrupt arrives
  • CPU saves context of P1, runs code to handle interrupt (e.g., read keyboard character)
  • Restore context of P1, resume P1

• **Interrupt handling code is part of OS**
  • CPU runs **interrupt handler of OS** and returns back to user code
Isolation

- How to protect processes from one another?
  - Can one process mess up the memory or files of another process?
- Modern CPUs have mechanisms for **isolation**
  - **Privileged** and **unprivileged** instructions
    - Privileged instruction = access to sensitive information (e.g., hardware)
    - Regular instructions (e.g., add) are unprivileged
- CPU has multiple **modes of operation** (Intel x86 CPUs run in 4 **rings**)
  - Low privilege level (e.g., ring 3) only allows **unprivileged instructions**
  - High privilege level (e.g., ring 0) allows **privileged instructions** also
- **User code** has unprivileged instructions, runs at **low privilege level**
  - CPU does not execute privileged instructions when in unprivileged user mode
- **OS code** has privileged instructions, runs at **high privilege level**
- When user program wants to do privileged operations, it must ask **OS**
  - CPU shifts to high privilege level, runs OS code, returns to low privilege, back to user code
CPU caches

- CPU must access memory to fetch instructions, load data into registers
  - But main memory (DRAM) is very slow (100s of CPU cycles)
  - CPU cannot do useful work while waiting for memory
- To avoid many memory accesses, CPU stores recently accessed instructions and data in CPU caches
  - Multi-level cache hierarchy, some private to cores, some common
  - Example: private L1, L2, common last level cache (LLC or L3)
  - Can be separate for instructions and data, or common (e.g., L1 is separate)
- Caches have low access latency (tens of CPU cycles), faster than DRAM but smaller in size, more expensive
  - Can only store most recently used instructions and data
Reading into cache

• Memory content fetched into cache in size of **cache line** (64 bytes)
  • When CPU requests contents at address X, 64 bytes around X are fetched
  • Why? Memory around recently accessed memory is most likely to be accessed in near future = **spatial locality of reference** (e.g., accessing array)

• Which level of cache hierarchy is data read into?
  • **Inclusive** cache: fetched into all levels of cache
  • **Exclusive** cache: fetched into lower level of cache

• What if cache is full?
  • **Least recently used (LRU)** cache lines are evicted into next level of cache
  • Why? Most recently used memory is most likely to be accessed again in near future = **temporal locality of reference** (e.g., for loop)
Writing into cache

- CPU writes into cached memory, makes it dirty
  - Dirty cache line = different from original copy in memory
- When is dirty cache line written back to memory?
  - Write through cache = written immediately
  - Write back cache = written later (more efficient)
- What if dirty cache line in private cache of one core needs to be accessed by another core?
  - CPU cores exchange modified data with each other
- Cache coherence protocols keep CPU caches in sync with each other and with main memory
Summary

• In today’s lecture:
  • Multiple CPU cores run programs in parallel ✓
  • Instructions, registers (ISA) ✓
  • Fetch-decode-execute ✓
  • Concurrent execution, interrupt handling, save/restore context ✓
  • Privilege levels ✓
  • CPU caches ✓

• Find information about your own CPU: which architecture? How many cores? What processes are running in your system?
  • Use Linux commands like “lscpu” and “top”