COMP124

Computer Systems

02 | CPU | Fetch-Execute Cycle | Instruction Sets

Dr Stuart Thomason

I/O Devices & Interrupts

- Devices include expansion cards (eg. graphics) that plug into the motherboard
- Modern motherboards also have some I/O devices and controllers built in
- Also includes peripheral (external) devices that plug in, such as keyboard, mouse, etc.
- When performing input/output, the CPU needs to know when a device is ready to receive or transmit, and when it has completed a request
- One option is to use polling
  - Periodically check the device status
  - Requires the CPU to stop what it's currently doing
  - Wastes time (CPU cycles)
- Modern computer systems use interrupts
  - Device sends a signal to the CPU when it's ready (or when it's finished)
  - Invokes an interrupt handler within the operating system
  - Intercepts the interrupt and decides when the CPU will handle it

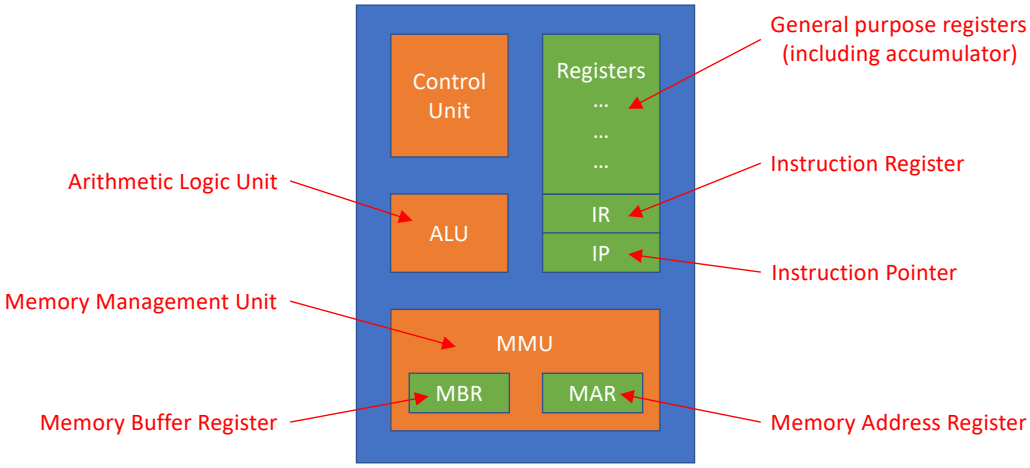
Internal Memory

- All programs and data must be converted to binary format and loaded into internal memory before they can be processed (the stored program concept)
- Memory can be…
  - RAM – Random Access Memory (read and write; volatile; main computer memory)
  - ROM – Read Only Memory (non-volatile; stores system boot code)
- The bit length of the system determines how much memory can be moved and manipulated by the CPU in one operation
- Most modern systems (desktops, laptops, phones) are 64-bit
- Many embedded microprocessors (inside appliances) are 8-bit or 16-bit
- It's possible to run a 32-bit operating system on a 64-bit system (many Windows installations use the 32-bit version even if the CPU is 64-bit)
- Software compiled for a 32-bit CPU can usually run on a 64-bit CPU (but not vice versa)

Bit Length & Word Size

- The bit length of the system is related to the word size of variables (data) in our code
  - Size of CPU registers
  - Width of the system bus
- The maximum unsigned integer that can be stored relates to the max memory size (because modern memory is byte addressable)
  - 16-bit – word – 64 kilobytes
  - 32-bit – dword – 4 gigabytes
  - 64-bit – qword – 16 exabytes
- Individual bits are zero-indexed from right to left
  - Bit zero (right-most) is called the least significant bit or lsb
  - Can also refer to least significant byte in the same manner

Inside the CPU

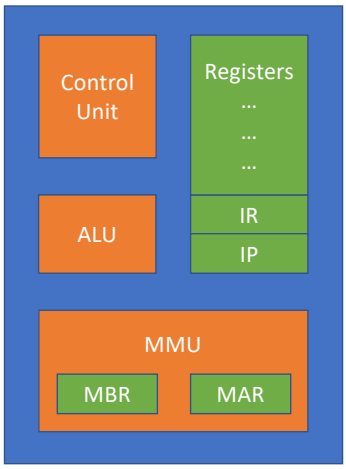


CPU Structure

- The activity of the CPU is governed by a complex piece of logic called the control unit
- The arithmetic logic unit (ALU) performs bit manipulations and numeric operations
- The control unit supplies the ALU with data (operands) and tells it what to do
- The CPU has internal data stores called registers
  - Access is much (much) faster than storing things in RAM
  - Have individual names and can be general purpose or have a specific use
  - Hold data temporarily while operations are being carried out by the ALU
- The instruction pointer (IP) always holds the address of the next instruction in memory (sometimes called the program counter, or PC)
- The instruction register holds the instruction currently being executed
- The memory address register and memory buffer register are used by the CPU to interface with main memory (and are not accessible to the programmer)

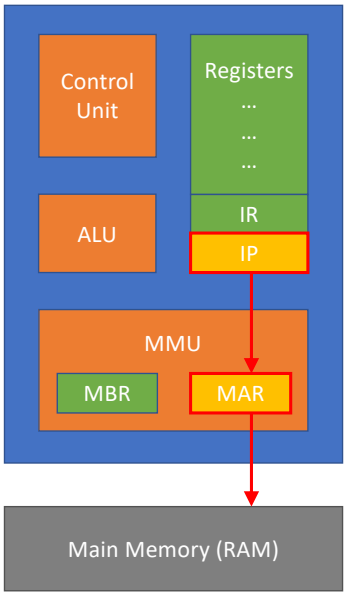
Fetch-Execute Cycle

- A compiled program is just a sequence of instructions in successive memory locations
- It will be loaded from disk into a contiguous chunk of memory
- To execute the program, the instruction pointer is set to point to the memory address of the first instruction



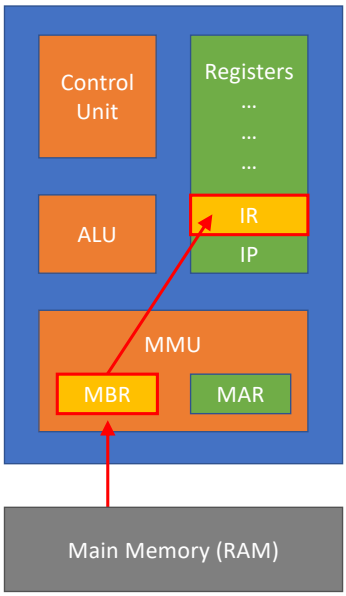
Fetch-Execute Cycle

- Step 1:
  - Copy address in IP into MAR
  - Issue read request to MMU
- Remember that memory access is slow (compared to the speed of the CPU)
- The CPU can get on with something else while it waits for the memory request to happen



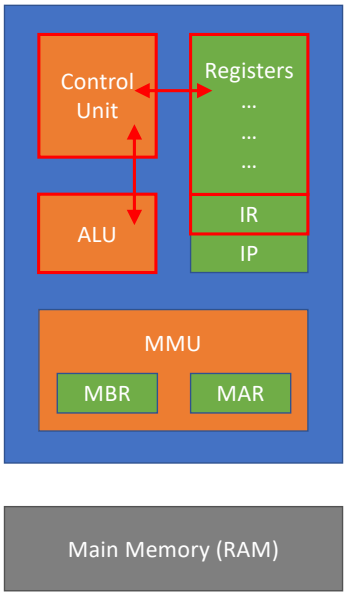
Fetch-Execute Cycle

- Step 2:
  - Increment IP to point to the next instruction
- Step 3:
  - Current instruction arrives from memory into MBR
  - Copy instruction into IR



Fetch-Execute Cycle

- Step 4:
  - Decode IR to work out what the instruction is
- Step 5:
  - Fetch any operands (extra data)
- Step 6:
  - Carry out the instruction (via ALU, etc.)
- Step 7:
  - Go to step 1
- Note that steps 5 and 6 might cause further memory access



Instruction Sets

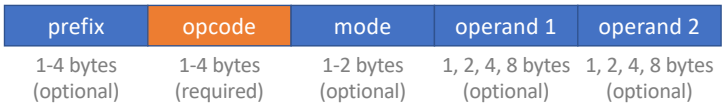
- The CPU instruction set provides various operations that fall into six broad categories
  - Transfer – moving data to and from memory and registers
  - Arithmetic – simple maths operations such as add, subtract, multiply, divide
  - Logic – bit manipulations such as AND, OR, NOT, shift, rotate
  - Test – comparing data values and setting status flags
  - Control – jumps and subroutine calls
  - Misc – various helper operations that don't fit into another category
- When you write code in a high level language (such as Java or C++) it needs to be compiled into a sequence of low level instructions that the CPU understands
- The fetch-execute cycle works its way through that low level sequence as it executes the program
- On this module we will write low level assembly code to give you an understanding of what is happening inside the CPU

Instruction Format

- Each instruction has an opcode that the CPU understands
  - For example, the number 5 might mean add
  - Whenever the CPU sees the number 5 stored in the instruction register, it knows it needs to perform an addition
- CPU gets further data (operands) from registers or main memory (via MMU requests)
- Many instructions have several opcodes because the operands can be encoded in different ways
  - Adding together the content of two registers
  - Adding a numeric value to a register
  - Adding data from a memory location to a register
- The CPU has to do something slightly different to get the data to be added
- So each opcode tells it exactly what to do and where to find the operands

Intel x86 CPU

- This module will use the Intel x86 (32-bit) instruction set
  - The 64-bit version is called x64
  - If you have issues in the labs, make sure you are compiling for x86 (not x64)



- A single x86 instruction can be anywhere from 1 to 15 bytes long
  - When the CPU sees the opcode in the instruction register, it might need to read more data from main memory to fully complete the instruction
  - The fetch-execute example was simplified to remove these extra steps
  - On this module we will assume every instruction takes up 4 bytes of memory (so we are bending the truth to make it easier to understand and explain)

Addressing Modes

- The mode part of the instruction tells the CPU where the operands are located
  - Immediate – operand value is encoded directly into the instruction (ie. a number)
  - Register – operand value is stored in a register
  - Direct – operand value is in main memory, so the instruction encodes its address
  - Register Indirect – instruction specifies a register that holds the address of the operand in main memory
- A fifth mode (implicit) is used when the instruction doesn't need an operand because it always does the same thing (eg. always manipulates or uses the same register)

Generating Instructions

- A compiler turns our high level code into a sequence of bytes that represent the opcodes and operands in the instruction set for the CPU we are targeting
- We will explore compilation in more detail towards the end of the module

- High level code to work out volume of sphere
- Compiled into sequence of bytes that CPU fetches
- Simple instructions need fewer bytes
- Note that this uses AT&T assembly syntax
- tinyurl.com/4k95y4u4

```
double sphere_volume(double radius)
{
    const double pi = acos(-1);
    return (4.0/3.0) * pi * pow(radius, 3);
}
```

```
Disassembly of section .text:

0000000000000000 <sphere_volume>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 83 ec 20           sub    $0x20,%rsp
   8: f2 0f 11 45 e8        movsd  %xmm0,-0x18(%rbp)
   d: f2 0f 10 05 00 00 00  movsd  0x0(%rip),%xmm0
  14: 00
  15: f2 0f 11 45 f8        movsd  %xmm0,-0x8(%rbp)
  1a: f2 0f 10 4d f8        movsd  -0x8(%rbp),%xmm1
  1f: f2 0f 10 05 00 00 00  movsd  0x0(%rip),%xmm0
  26: 00
  27: f2 0f 59 c8           mulsd  %xmm0,%xmm1
  2b: f2 0f 11 4d e0        movsd  %xmm1,-0x20(%rbp)
  30: f2 0f 10 05 00 00 00  movsd  0x0(%rip),%xmm0
  37: 00
  38: 48 8b 45 e8           mov    -0x18(%rbp),%rax
  3c: 66 0f 28 c8           movapd %xmm0,%xmm1
  40: 66 48 0f 6e c0        movq   %rax,%xmm0
  45: e8 00 00 00 00        callq  4a <sphere_volume+0x4a>
  4a: f2 0f 59 45 e0        mulsd  -0x20(%rbp),%xmm0
  4f: 66 48 0f 7e c0        movq   %xmm0,%rax
  54: 66 48 0f 6e c0        movq   %rax,%xmm0
  59: c9                    leaveq
  5a: c3                    retq
```