5.6. The Instruction Cycle

5.6.1. Generic CPU Instruction Cycle

The generic instruction cycle for an unspecified CPU consists of the following stages:

1. Fetch instruction: Read instruction code from address in PC and place in IR. (IR ← Memory[PC])
2. Decode instruction: Hardware determines what the opcode/function is, and determines which registers or memory addresses contain the operands.
3. Fetch operands from memory if necessary: If any operands are memory addresses, initiate memory read cycles to read them into CPU registers. If an operand is in memory, not a register, then the memory address of the operand is known as the *effective address*, or EA for short. The fetching of an operand can therefore be denoted as Register ← Memory[EA]. On today's computers, CPUs are much faster than memory, so operand fetching usually takes multiple CPU clock cycles to complete.
4. Execute: Perform the function of the instruction. If arithmetic or logic instruction, utilize the ALU circuits to carry out the operation on data in registers. This is the only stage of the instruction cycle that is useful from the perspective of the end user. Everything else is overhead required to make the execute stage happen. One of the major goals of CPU design is to eliminate overhead, and spend a higher percentage of the time in the execute stage. Details on how this is achieved is a topic for a hardware-focused course in computer architecture.
5. Store result in memory if necessary: If destination is a memory address, initiate a memory write cycle to transfer the result from the CPU to memory. Depending on the situation, the CPU may or may not have to wait until this operation completes. If the next instruction does not need to access the memory chip where the result is stored, it can proceed with the next instruction while the memory unit is carrying out the write operation.

An example of a full instruction cycle is provided by the following VAX instruction, which uses memory addresses for all three operands.

```
mull    x, y, product
```

1. Fetch the instruction code from Memory[PC]
2. Decode the instruction. This reveals that it's a multiply instruction, and that the operands are memory locations `x`, `y`, and `product`.
3. Fetch `x` and `y` from memory.

4. Multiply `x` and `y`, storing the result in a CPU register.
5. Save the result from the CPU to memory location `product`.

5.6.2. MIPS Instruction Cycle

Since the MIPS is a load-store architecture, all instructions except load and store get their operands from CPU registers and store their result in a CPU register. Hence, the instruction cycle for all instructions except load and store is somewhat simpler. When all operands are in CPU registers, which can be accessed within a single clock cycle, fetching operands and storing the results can occur within the same clock cycle as execution (add, subtract, etc.). For example, suppose R0, R1, R2 ... R15 are CPU registers. Then the operation

```
R0 ← R4 + R7           # One clock cycle
```

is a simple, atomic operation inside the CPU, and therefore is not regarded as multiple steps in the instruction cycle. If one of operands were in memory instead of a register, on the other hand, fetching it from memory and placing it into a register would be a separate step.

```
R4 ← Mem[address1]     # Multiple clock cycles
R0 ← R4 + R7           # One clock cycle
```

1. Fetch instruction from memory to IR
2. Decode
3. Execute (all data in CPU registers)

The specific cycle for a load instruction is:

1. Fetch instruction from memory to IR
2. Decode
3. Fetch operand from memory to a register

The specific cycle for a store instruction is:

1. Fetch instruction from memory to IR
2. Decode
3. Store operand from register to memory

5.6.3. Analysis of the Instruction Cycle

Note that in any case, most of the instruction cycle is overhead. Only the execute stage actually does something considered useful by the user, and all the other stages are fluff, either preparation or wrap-up.

One way to increase the density of useful work in a program is by making more complex instructions. If the execute cycle accomplishes more for the same amount of fetching, decoding and storing overhead, then the program will be shorter, and will run faster. This is the philosophy behind CISC architectures. A classic example of this idea is the VAX `polyf` instruction, which evaluates a polynomial given an array of coefficients, the order or the polynomial, and the value of x. It accomplishes in one instruction cycle what would require a loop, and hence dozens of instruction cycles otherwise.

The cost of overhead can also be alleviated without actually reducing it. The primary technique to achieve this is called *pipelining*. A pipelined CPU overlaps the execution of two or more instructions, so that while one instruction is executing, the next one is already being decoded, and the one after that is being fetched.