ENEE646 Fall 2008 Midterm Answers

October 30, 2008

Problem 1

Question

COMPUTER OPERATION. Transistors make up logic gates that execute a computer program, and a program is the physical realization of an algorithm or concept. Project I filled in the gap between the two extremes: it dealt with the translation from human-readable code (i.e., an algorithm) to machine-readable code and the translation from machine-readable code to execution of that code.

So explain the process: how do you get from human-readable code (given that assembly code is at least *vaguely* human-readable) to something a computer understands, and how does that machine-specific version of the algorithm get executed? In short, how does a computer system work?

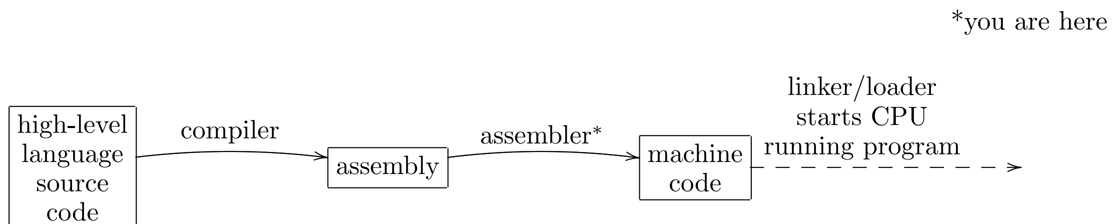
Answer

Translating from assembly to machine code:

Ignoring assembly pseudo-operations (where one assembly directive can translate to multiple instructions or data words), the vast majority of assembly instructions map directly to binary machine instructions. For ease of use, assembly code typically includes labels (a.k.a. symbols) to identify code/data. So the assembler must:

1. make a pass through the code to identify symbol locations (build a symbol table)
2. make a pass, translating each human-readable instruction into a machine instruction, filling in symbol values/offsets where necessary (writes out binary machine code containing opcode, operands, and immediates)

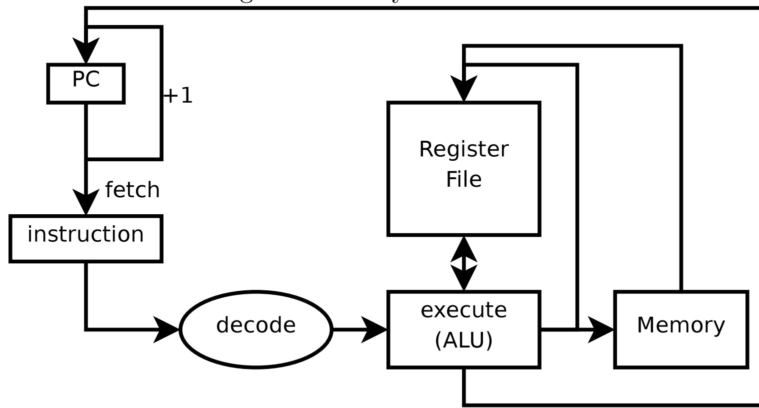
This fits into an overall development system as follows:



Running the machine code:

The CPU is a state machine with input, output, control, and memory. The state (memory, registers) is in sequential logic; combinational logic in the form of control logic decides what to do. Every clock, we may fetch an instruction, decode it (read the opcode to decide what to do), execute it, and then update the state

of the machine. Diagrammatically:



Problem 2

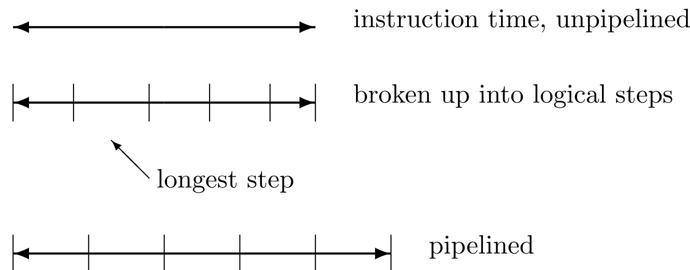
Question

Discuss the technique of **PIPELINING** a processor design. What are the tricky implementation issues? What is the effect of pipelining on processor performance? (support your answer quantitatively)

Extra credit: what is the effect of pipelining on power dissipation? (support your answer quantitatively)

Answer

Pipelining increases throughput but at the expense of making the overall time to execute a single instruction longer. This is because we break up the execution into logical steps; one step executes per clock cycle. The clock speed is therefore determined by the longest step:



Overall, execution time is $T \cdot I \cdot C$:

$$= \text{Time per clock cycle} \times \text{Instructions} \times \#\ \text{cycles per instruction}$$

Implementation of pipelining adds complexity:

- pipeline registers must be added to hold all the necessary data for each step and pass information (after doing some work) onto the next step
- control logic must happen per-step
- to avoid breaking the contract with the programmer, we must detect:
  - hazards e.g. load-use hazards or ALU reuse hazards or write-read hazards

    - write-read hazards can be mitigated by forwarding data from later in pipe – closest step best!
  - branch (mispredict) penalties
    - need to cleanup undesired instructions in this case (e.g. replaces with NOPs)
    - easier if we avoid committing state until end of pipe

Pipelining also increases power dissipation, especially if we increase clock speed:

$$Power \approx cv^2 f$$

But actually, $Power = c \cdot V_{dd} \cdot \Delta V \cdot \alpha f + leakage_current$
Even if we do not change f, α increases with complexity (more components/activity) as does leakage current.