CS1101: Lecture 28

The Microarchitecture Level

Dr. Barry O'Sullivan
b.osullivan@cs.ucc.ie

Course Homepage
http://www.cs.ucc.ie/~osullb/cs1101

Lecture Outline

- What is a microarchitecture?
- OPCODE
- Stacks
- How does a stack works?
- Use of a stack for storing local variables
- Operand Stacks
- Use of an Operand Stack
- **Reading**: Tanenbaum, Chapter 4: 4.1, 4.2, 4.5.1

Where are we now?

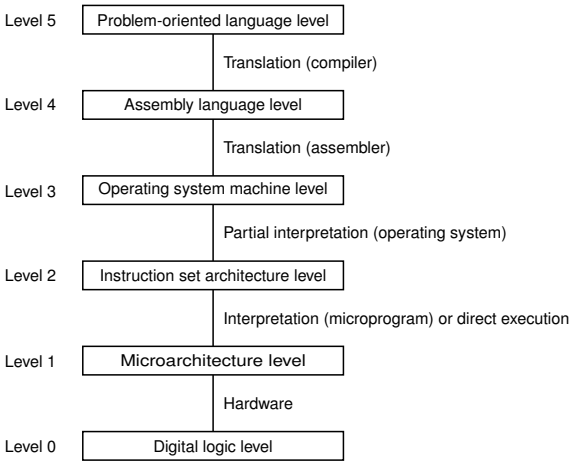

Figure 1-2. Our six-level computer.

Introduction

- The level above the digital logic level is the **microarchitecture level**.
- Its job is to implement the ISA (Instruction Set Architecture) level above it, as illustrated in Fig. 1-2.
- The design of the microarchitecture level depends on the ISA being implemented, as well as the cost and performance goals of the computer.
- Many modern ISAs, particularly RISC designs, have simple instructions that can usually be executed in a single clock cycle.
- More complex ISAs may require many cycles to execute a single instruction.
- Executing an instruction may require locating the operands in memory, reading them, and storing results back into memory.

What is a microarchitecture?

- A convenient model for the design of the microarchitecture is to think of the design as a programming problem, where each instruction at the ISA level is a function to be called by a master program.
- In this model, the master program is a simple, endless loop that determines a function to be invoked, calls the function, then starts over.
- The **microprogram** has a set of variables, called the **state** of the computer, which can be accessed by all the functions.
- Each function changes at least some of the variables making up the state.
- For example, the Program Counter is part of the state – the memory location for the next instruction.

OPCODE

- Each instruction has a few fields, usually one or two, each of which has some specific purpose.
- The first field of every instruction is the **opcode** (short for **operation code**).
- The opcode identifies the instruction, telling whether it is an ADD or a BRANCH, or something else.
- Many instructions have an additional field, which specifies the **operand**.
- For example, instructions that access a local variable need a field to tell which variable.
- This model of execution is sometimes called the **fetch-execute cycle**.

Stacks

- Virtually all programming languages support the concept of procedures (methods), which have local variables.
- These variables can be accessed from inside the procedure but cease to be accessible once the procedure has returned.
- Where should these variables be kept in memory?
- The simplest solution, to give each variable an absolute memory address, does not work, since a procedure may call itself (be recursive).
- Why?
- If a procedure is active (i.e., called) twice, it is impossible to store its variables in absolute memory locations because the second invocation will interfere with the first.

How does a stack works?

- Instead, an area of memory, called the **stack**, is reserved for variables, but individual variables do not get absolute addresses in it.
- Instead, a register, say, LV, is set to point to the base of the local variables for the current procedure.
- Another register, SP, points to the highest word of the local variables.
- Variables are referred to by giving their offset (distance) from LV.
- The data structure between LV and SP (and including both words pointed to) is called a variable's **local variable frame**.

Use of a stack for storing local variables

Figure 4-8. Use of a stack for storing local variables. (a) While A is active. (b) After A calls B. (c) After B calls C. (d) After C and B return and A calls D.

Operand Stacks

- Stacks have another use, in addition to holding local variables.
- They can be used for holding operands during the computation of an arithmetic expression.
- When used this way, the stack is referred to as the **operand stack**.
- Example: suppose that before calling B, A has to do the computation:

$$a1 = a2 + a3$$

Use of an Operand Stack

- Push a2 onto the stack – here SP has been incremented by the number of bytes in a word, say, 4, and the first operand stored at the address now pointed to by SP.
- Next, a3 is pushed onto the stack.
- The actual computation can be done by now executing an instruction that pops two words off the stack, adds them together, and pushes the result back onto the stack.
- Finally, the top word can be popped off the stack and stored back in local variable a1.

Use of an Operand Stack

Figure 4-9. Use of an operand stack for doing an arithmetic computation.