Disclaimer: The contents of this document are scribe notes for The University of Texas at Austin EE360N Fall 2008, Computer Architecture. The notes capture the class discussion and may contain erroneous and unverified information and comments.

Basics of ISA Implementation

Lecture #2: 09/03/2008
Lecturer: Derek Chiou
Scribe: Manish Arora, Gavin Rade, Gage Eads

1 Recap of Last Lecture and Outline of This Lecture

Computer Architecture is the art of tradeoffs. Computer architecture deals with optimizing under constraints such as electrical power consumption, development costs, per part cost, verification effort and performance amongst other factors. The Instruction Set Architecture (ISA) is the interface between the programmer (or compiler) and the hardware. The ISA differs from the micro-architecture of the computer, which is an optimal implementation of the ISA. The example we had used was cooking the Chicken with Peanuts (CWP) recipe (our computing task) using bowls, cutting board, wok, etc. (our hardware resources) by performing basic operations such moving, cutting and control (our ISA) on our ingredients (our input data to be processed).

This lecture builds on previous concepts by introducing the memory and storage. To aid in the understanding of concepts we briefly provide analogies to storage used in our CWP cooking example and real computer systems. We then move on to explaining rationale and evolution of ISAs and followed by an introduction to micro-programmed architectures.

1.1 Storage

- Storage is necessary for values needed in the future. Just like our CWP recipe can use a combination of a refrigerator, or bowls or the wok to save the chicken (data), computers use a combination of memory storage methods such as the cache, hard disk or RAM. During the process of storage there are important issues to consider:

1.1.1 Size and Latency

– Bigger size might seem good at first but we would also have to deal with the extra latency, cost and power consumption associated with the bigger size. Hard disk drives have large amounts of storage, but are much slower than RAM. A bigger kitchen is nice because it stores more, but takes longer to move around in.

1.1.2 Bandwidth

– Higher bandwidth seems better but the costs associated with having this are significant.

1.1.3 Granularity

– It is important to consider whether we use the same sizes for our memory storage elements or different sizes. Having same sizes is convenient when your operations required multiple storages of the same size, but some operations can be optimized when your storage locations vary in size.

1.1.4 Naming

– It is very difficult for computers to process names the way humans use them. We use names with variable length and use rules that are apparent to us or formed in context which are not easy to program in a computer. How would a computer know the end of a name when some have two words and some names have three? Numerical addresses solve this issue. Phone numbers or social security numbers use fixed 9 digit names, which are easily interpreted. Numerical addresses are also easily mapped across ranges or can be mapped to smaller sets or addresses in memory. Computers also use symbolic names (such as variable names in programs) to represent data. Because such symbolic names have rules governing them they can be easily mapped to a unique address or range of addresses by an automated system such as a compiler.

2 Historical Architectures

Early machines were focused on minimizing everything. In the early days of computing the building blocks of logic were vacuum tubes, which were both expensive and unreliable. Computers also suffered from having only hundreds or thousands of words of memory. Given limited and expensive resources, architects were forced to minimize the size of all aspects of architecture. For example, instruction sizes were as small as possible,

minimal memory/registers were available, ALU sizes and functions were matched to 1 bit size to the application requirements and the use of parallelism was rare. In fact, one machine had 17-bit ALUs - because 16 bits was insufficiently small, and 18 too big. For example, the LC3-b doesn't have a subtract instruction, because you can merely negate a number then add it.

3 Instruction Set Architectures, Salient Features and Considerations

There are a number of alternatives available on how we choose to implement instruction set architectures. Instructions can be implemented in hardware, such as a hardware square root instruction (at the cost of resources). Array processors are an example, but they are very difficult to program.

3.1 Static Instruction Set Architectures

- Having a static ISA simplifies the hardware/software interface. Since software explicitly knows the ISA its easy to write and hardware can be optimally implemented. The ISA specification provides a way for software and hardware compatibility. This abstraction protects the software and hardware, because developing/updating one will not interfere with the other. Since the ISA does not change on new hardware we can run old software and we can run new software on old machines.

3.2 Sequential Semantics

- Most ISAs have sequential semantics, i.e., instructions are executed in order one at a time. The result of the present instruction is available before the next instruction executes. This makes it easy to understand and reason about programs. At the same time each instruction can change machine states. The next instruction starts with the new state as its initial state configuration.

3.3 Number of Instructions in an ISA

- The number of instructions in an ISA is a tradeoff between the ease of implementing instructions and the difficulty the programmer would face in implementing the algorithm.

  subleq a, b, c ; Mem[b] = Mem[b] - Mem[a] ;if (Mem[b] = 0) goto c

  For example single instruction ISAs such as the one below can implement all possible operations and are easy to implement in hardware but would be difficult to program and yield huge program sizes.

3.4 Number of Registers per Instruction

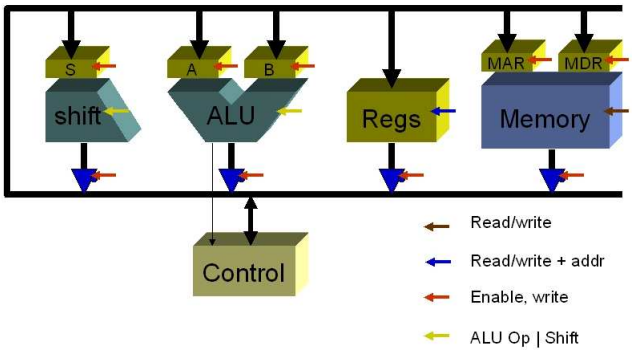
- ISA instructions and their operands need to be encoded into machine code for program storage. The number of registers that can be specified as arguments to an instruction varies and has implications in how many bits are utilized for register representation, memory access or immediate value specifications in instructions. The number of registers used varies from zero for stack machines, one for accumulator-based machines, two for instructions that use a source/destination (source can be memory) and three for instructions that specifies two registers as arguments and a third as a result storage. Four or more registers are not commonly used and require larger instructions.

3.5 Variable Length Instructions

- If we fix the length of instructions we cannot encode instructions with a dynamic (potentially long) range, which could utilize a lot of registers or long immediate values. At the same time we might not be able to utilize all the available bits for smaller instructions. Variable length instructions provide a more compact way to encode instructions to the exact length needed. We also get the advantage of being able to extend or add new instructions. Finite state machines (FSMs) can be easily used to interpret variable length instructions. Decoding these instructions is just a matter of parsing, but walking through each byte in a long instruction may slow the computer down.

4 Implementing ISAs using Microcode

ISAs can be implemented by finite state machines that control individual components of a processor. Control is encoded in microcode, which is how the machine goes cycle by cycle through each instruction. Each state of the FSM implements one step of the instruction. Steps consist of controlling the components of the processor to implement the instruction. Microcode can itself be coded further to save on microcode space. The figure below shows an example of micro coded computer's data path. The various functional units are connected via a bus, that is used by the control unit and functional units, to exchange data. The data path consists of a number of functional units that may or may not be exposed to the programmer but are useful in implementing the micro-coded ISA.



The control hardware sends signals along dedicated wires (different than the bus), to the various structures to control them. The control is directly linked to the ALU to receive the nzp bits necessary to its function. Micro-coded machines are efficient because they have the capabilities to reuse components and different parts of the micro-architecture in a programmable and efficient manner. They also use simple components, and a single ALU which is sufficient to perform address operations.

5 Evolution of CISC Processors and ISA Explosion

Microcode can make a more complex instruction more efficient to implement than multiple simpler instructions that perform the same task. Consider the example of a CISC instruction below, which uses about 8 microcode operations for execution.

ADD DR, SR1, SR2, SR3 $\{DR = SR1 + SR2 + SR3\}$

This microcode this instruction uses is:

Regs[SR1]$\rightarrow A$;
Regs[SR2]$\rightarrow B$;
ADD$\rightarrow ALU$;
ALU$\rightarrow A$;
Regs[SR3]$\rightarrow B$;
ADD$\rightarrow ALU$;
ALU$\rightarrow Regs[DR]$;

The instruction when implemented on a simpler machine might be in the form of 2 instructions, would require 2 instruction stores, an extra register and about 12 micro-code operations.

ADD DR1, SR1, SR2
ADD DR, SR3, DR1

The complex instruction has a much efficient micro-code implementation without any additional hardware. Coupled with the availability of cheap ROM to store large microcode there has been a rapid growth and introduction of complex ISAs. With Microcoding instructions are easy to implement at no further hardware cost. This led to a large accumulation of (sometimes unnecessary) instructions, such as the VAX polynomial instruction. At times the instruction growth has been uncontrolled, which leads to backwards compatability difficulties in the futre.

6 Memory Addressing Modes

As another example of the ISA explosion lets consider the growth in the support of memory addressing modes in instructions. We typically have direct addressing (the address is fully encoded in the instruction), register indirect addressing (register carries the address), indexed addressing (register along with an offset creates the address), base indexed addressing (two registers along with an offset create the address) and auto increment addressing (post modification of address registers).

7 ISA, Micro Architecture and Compiler Interaction

Since the micro-architecture of a processor determines the hardware resources available to implement the ISA, the ISA is dependent on the microarchitecture implementation and needs to be suitably optimized. For example if we need to implement a 4 register opcode instruction on a pipelined processor we need to have a register file with 4 read ports. Unless the micro architecture supports such a read capability the 4-register opcode instruction will take multiple cycles to read from the register file.. Even if the micro-architecture supports the 4 read ports it has to do so at a steep hardware cost. Having an extra read port has a high cost, since the area of a register file is proportional to the square of the number of ports; a four-ported register file is 16 units of area compared to 9 units of area for one three ported register file.

The compiler's job is to interpret and assemble code. In the early days complier performance was poor and instructions were easy to add leading to complex ISAs. In time, compliers have become more powerful and new ISAs balance between compiler efficiency and ease of implementation.