Overview

The ISA is defined as how the machine appears to a (machine level) programmer, or more importantly, the compiler

In order to write the compiler, we have to know what addressing modes, what registers, and what data types and instructions are available

Let's first consider the taxonomy of Instruction Set Architectures (ISA)

Taxonomy of ISAs:

Stack Architecture:

Operands are implicitly on the top of the stack

Accumulator Architecture:

One operand is implicitly in the 'accumulator register'

Taxonomy of ISAs

General Purpose Register (GPR) Architecture: Three general types:

- **Memory-memory**:
  May have 2 or 3 operands in memory (old VAX model)

- **Register-memory:**
  Operations occur between register and memory -- usually 2 operands, one in a register (src and dest) and one in memory (src only)

- **Load-store**:
  Data must be explicitly moved between registers and memory
  ALU operations use register operands only
  Usually 3 operands, all in registers.

Stack	Accum	Mem-mem	Reg-mem	Reg-reg
Push A	Load A	Add C, A, B	Load R1, A	Load R1, A
Push B	Add B		Add R1, B	Load R2, B
Add	Store C		Store C, R1	Add R3, R1, R2
Pop C				Store C, R3

Taxonomy of ISAs

Why are GPR ISAs so popular?

- Registers are MUCH faster than memory
- Compiler can use them effectively to:
  - Evaluate expressions
  - Hold variables
  - Pass parameters

Two instruction set characteristics divide GPRs:

The number of ALU operands (2 or 3)

The number of operands that may be memory addresses (0-3)

ISA Metrics

- *Instruction density*:
  How much space does a program require?
  Mem-mem and Reg-mem: good instruction density

- *Instruction count*:
  How many instructions are necessary for a specific task?
  Reg-reg usually have large instruction counts compared to Mem-mem and Reg-mem

- *Instruction complexity*:
  How much decoding is necessary to interpret an instruction?
  Mem-mem most complex, Reg-reg simplest

- *Instruction length*:
  Is length dependent on the type of instruction and addressing mode?
  Mem-mem instruction length can be variable and usually longer than Reg-reg due to memory operands
  Reg-reg instructions are usually fixed in length

Memory addressing

How is a memory address interpreted?

What byte is specified by the address ?

Two conventions for byte ordering:

- **Big Endian**: The address of the "word" is the address of the most significant (biggest) byte
- **Little Endian**: The address of the "word" is the address of the least significant (littlest) byte

Problem: When word (binary) data is transferred between the two types of machines, byte swapping is necessary

Alignment

On many machines, accesses to objects larger than a byte must be aligned. i.e.

A 4-byte integer must be stored at an address divisible by 4 for word alignment

Why?

Misalignment complicates the hardware

Addressing Modes

An addressing mode can specify a constant, a register or a location in memory

- *Register*: Operand is in a register
  Add R4, R1
- *Immediate*: Operand is a constant encoded directly in the instruction
  Add R4, #10
- *Displacement*: Memory address is computed by adding a constant (found in the instruction) to the value in the register
  Add R4, 100(R1)
- *Indirect*: Similar to displacement except the constant is 0
  Add R4, (R1)
- *Indexed*: Two registers are added together to get the memory address
  Add R3, (R1 + R2)

Addressing Modes

- *Direct/Absolute*: The memory address is contained directly in the instruction

  Add R1, (1001)

- *Auto-increment/decrement*: The value in a register is either incremented/decremented, either before or after the register's value is used as a memory address

  Add R1, (R2)+

  Add R1, -(R2)

- *Scaling*: Similar to *indexed* except that the index value may be implicitly multiplied by a constant (2, 4 or 8) in order to access halfwords, words or doublewords. Also, it contains an explicit constant

  Add R1, 100(R2)[R3]

- *Memory deferred*: Allows additional levels of indirection

  Add R1, @(R3)

Addressing Modes

We skipped *PC-relative* addressing modes for the moment

These specify code addresses in control transfer instructions

Addressing mode impact:

- Can significantly reduce instruction count
- Add complexity to the hardware
- May increase the average number of clocks per instruction (CPI)

Addressing Modes

Important addressing modes:

- **Register**: Provides the means to use the registers

- **Displacement**: Provides the means of implementing pointers

  Add R4, 100(R1) where R1 holds the address of a data item

  Issue: What is the appropriate displacement field size?

  Important because it affects instruction length

  Program analysis shows that:

  12 bits capture ~75% of full 32-bit displacements found in programs

  16 bits capture ~99% of full 32-bit displacements found in programs

  Therefore, 12-16 bits is probably sufficient

Addressing Modes

Important addressing modes:

- **Immediate**: Used in arithmetic operations (comparisons) and in moves where a constant is needed in a register

Issue: What is the appropriate immediate field size?

Important because it affects instruction length

Program analysis shows that:

Most immediate values are less than 8 bits

However, large immediates are sometimes used, most often in address calculations

Therefore, 8-16 bits is probably sufficient

Integer programs use immediates quite often (up to 1/3 of all instructions) while floating point programs use them less often (1/10)

Other addressing modes are certainly useful, but are they worth the chip space and design complexity?

Instruction Set Operations

Arithmetic/Logical: Integer ALU ops

ADD, **AND**, **SUB**, **OR**

Load/Stores: Data transfer between memory and registers

LOAD, **STORE** (Reg-reg), **MOVE** (Mem-mem)

Control: Instructions to change the program execution sequence

BEQZ, **BNEQ**, **JMP**, **CALL**, **RETURN**, **TRAP**

System: OS instructions, virtual memory management instructions

INT

Floating Point:

FADD, **FMULT**

Decimal: Support for BCD
String: Special instruction optimized for handling ASCII character strings
Graphics: Pixel operations, compression and decompression

Instruction Set Operations

All machines generally provide a full set of operations for the first three categories

All machines MUST provide instruction support for basic system functions

Floating point instructions are optional but are commonly provided

Decimal and string instructions are optional but are disappearing in recent ISAs

- They can be easily emulated by sequences of simpler instructions

Graphic instructions are optional

Remember **MAKE THE COMMON CASE FAST**?

- ALU and Load/Store instructions represent a significant portion of the instruction mix and therefore should execute quickly

Instruction Set Operations

Control Flow instructions:

Four types are identifiable:

- Conditional branches
- Jumps
- Procedure Calls
- Procedure Returns

Program analysis shows that Conditional branches dominate (> *80%*)

The destination address must always be specified

In most cases, it is given explicitly in the instruction

Control Flow Operations

Addressing Modes:

PC-relative: Most common

Constant in instruction gives the offset to be added to the PC

Adv:

Since target is often near the current instruction, the displacement can be small, requiring few address bits

Allows relocatable (position independent) code

Indirect (jump to the address given by a register)

For procedure returns and indirect jumps for which the address is not known at compile time

Register indirect jumps useful for three important features:

- Case or switch statements
- Dynamic shared libraries
- Virtual functions

Absolute (jump to location in memory) -- not commonly used

Control Flow Operations

Conditional branches:

Issue: What is the appropriate **field size** for the offset?

Important because it affects instruction length and encoding

Observations:

- Most frequent branches for integer programs are targets 4 to 7 instructions away (for DLX). This suggests a small offset field is sufficient

- Most **non-loop** branches (up to *75%* of all branches) are forward

  However, they are hard to "predict" and optimize

- Most **loop** branches are backward

  Backward branches are usually taken, since they are usually part of loops

Control Flow Operations

Conditional branches: Methods of testing the condition:

- *Condition Codes (CC)*:

  Special bits are set by ALU operations as a side effect

  Adv:

  Reduces instruction count - it's done for free

  Disadv:

  Extra state that must be implemented

  More importantly, it *constrains* the ordering of instructions (no intervening instructions allowed (that set the CC) between the instruction that sets the CC and the branch that tests the CC)

- *Condition register*:

  Comparisons leave result in a register, which is tested later

Subroutines:

Include control transfer and return + some state saving

Encoding an Instruction Set

An architect more interested in code size will pick *variable* encoding

- Allows virtually all addressing modes in all operations
- This style is best when there are lots of addressing modes and operations
- Instruction differ significantly in the amount of work performed

An architect more interested in simplifying instruction decoding in the CPU will pick *fixed* encoding

- Operation and addressing mode encoded into the opcode

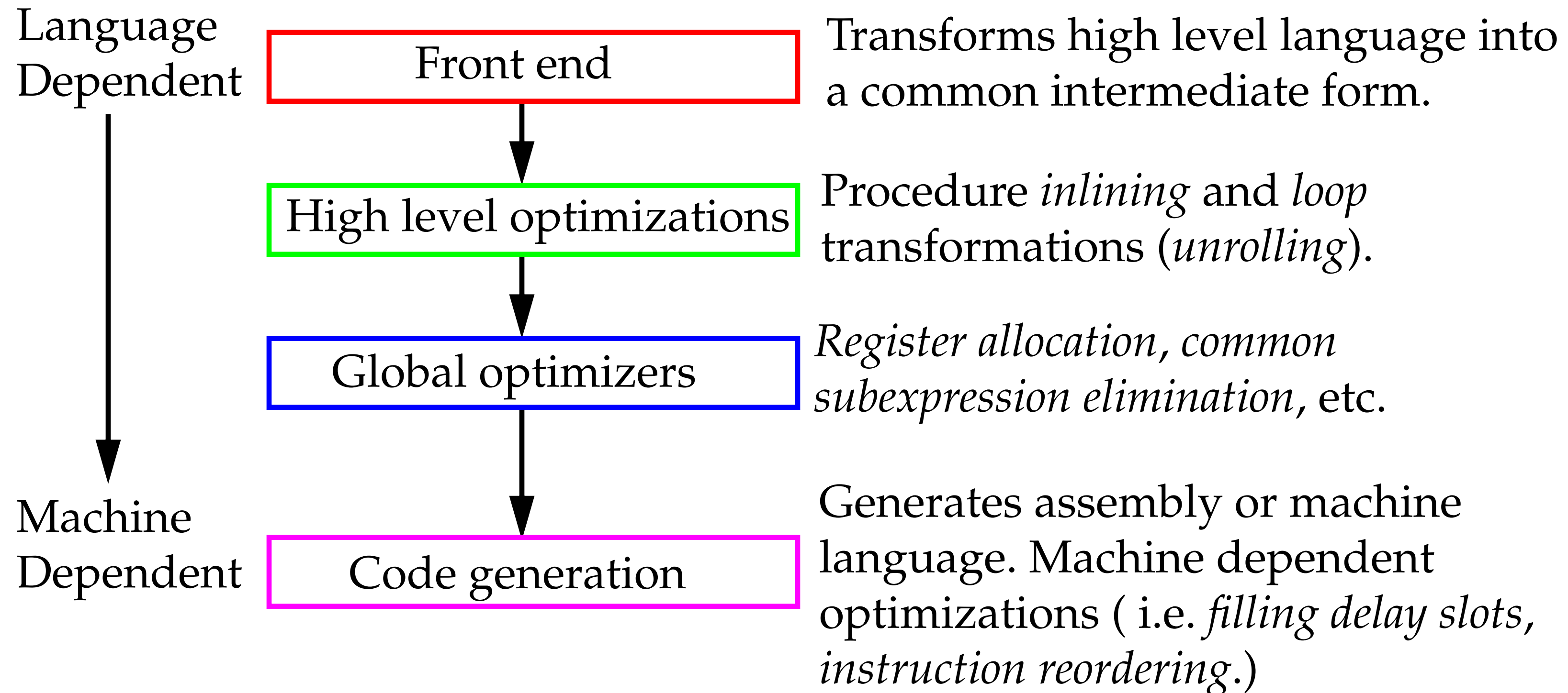
Compilers and Architecture

What features of an architecture lead to high quality code?

What “makes it easy” to write efficient compilers for an architecture ?

Compilers and Architecture

Structure of optimizing compilers:



Instruction set properties that help the compiler writer:

- *Regularity*:

  The 3 primary components of an instruction set; **operations**, **data types** and **addressing modes** should be *orthogonal* (independent).

Compilers and Architecture

- *Regularity*: (continued)

  Consider operations and addressing modes:

  They are orthogonal if any address mode can be used for any operation

  No dependencies, i.e. MOV instruction applicable only to register mode or even worse, to only a subset of the registers

- *Provide primitives, not solutions*:

  Providing special features that "match" language constructs is NOT a good idea

  These features may be good only for a certain language

  And, worse, they may match but do more or less than what's required

- *Simplify trade-offs among alternatives*:

  If there are 20 ways to implement an instruction sequence, it makes it difficult for the compiler writer to choose which is the most efficient