

Instruction Set Architecture

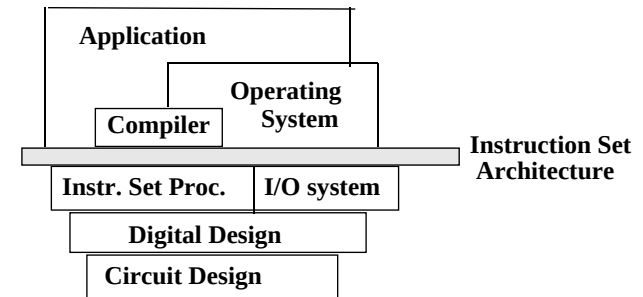
or

“How to talk to computers if
you aren’t on Star Trek”

CSE 240

Dean Tullsen

The Instruction Set Architecture



CSE 240

Dean Tullsen

Crafting an ISA

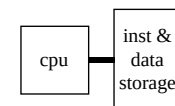
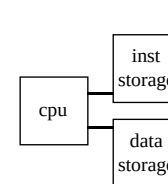
- Designing an ISA is both an art and a science
- ISA design involves dealing in an extremely rare resource -- instruction bits!
- Some things we want out of our ISA
 - completeness
 - orthogonality
 - regularity and simplicity
 - compactness
 - ease of programming
 - ease of implementation

CSE 240

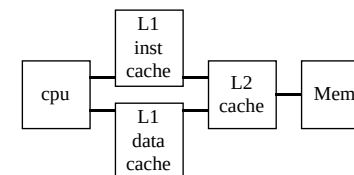
Dean Tullsen

Where are the instructions?

- Harvard architecture
- Von Neumann architecture



“stored-program” computer

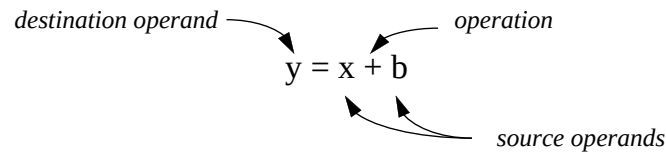


CSE 240

Dean Tullsen

Key ISA decisions

- operations
 - how many?
 - which ones
- operands
 - how many?
 - location
 - types
 - how to specify?
- instruction format
 - size
 - how many formats?



how does the computer know what
0001 0100 1101 1111
means?

CSE 240

Dean Tullsen

Choice 1: Operand Location

- Accumulator
 - Stack
 - Registers
 - Memory
- We can classify most machines into 4 types: *accumulator*, *stack*, *register-memory* (most operands can be registers or memory), *load-store* (arithmetic operations must have register operands).

CSE 240

Dean Tullsen

Choice 1B: How Many Operands?

Basic ISA Classes

Accumulator:

1 address add A $acc \leftarrow acc + mem[A]$

Stack:

0 address add $tos \leftarrow tos + next$

General Purpose Register:

2 address add A B $EA(A) \leftarrow EA(A) + EA(B)$

3 address add A B C $EA(A) \leftarrow EA(B) + EA(C)$

Load/Store:

3 address add Ra Rb Rc $Ra \leftarrow Rb + Rc$
 load Ra Rb $Ra \leftarrow mem[Rb]$
 store Ra Rb $mem[Rb] \leftarrow Ra$

A *load/store* architecture has instructions that do either ALU operations or access memory, but never both.

CSE 240

Dean Tullsen

Alternative ISA's

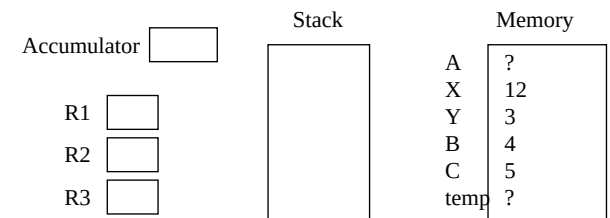
- $A = X * Y - B * C$

Stack Architecture

Accumulator

GPR

GPR (Load-store)



CSE 240

Dean Tullsen

Trade-offs

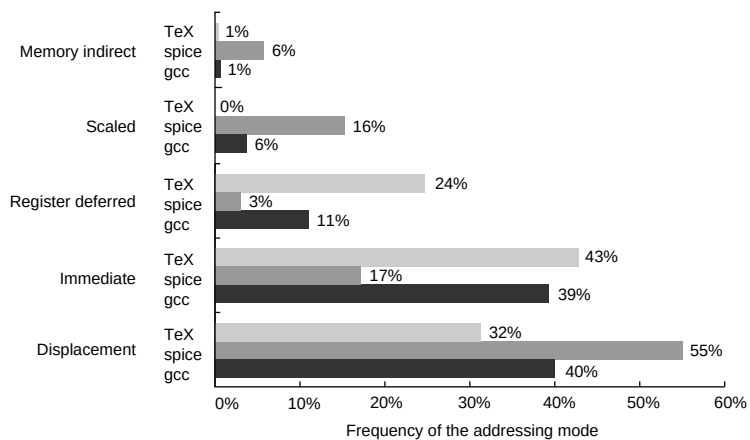
Stack	
+	-
Accumulator	
+	-
GPR	
+	-
Load-store	
+	-

Choice 2: Addressing Modes

how do we specify the operand we want?

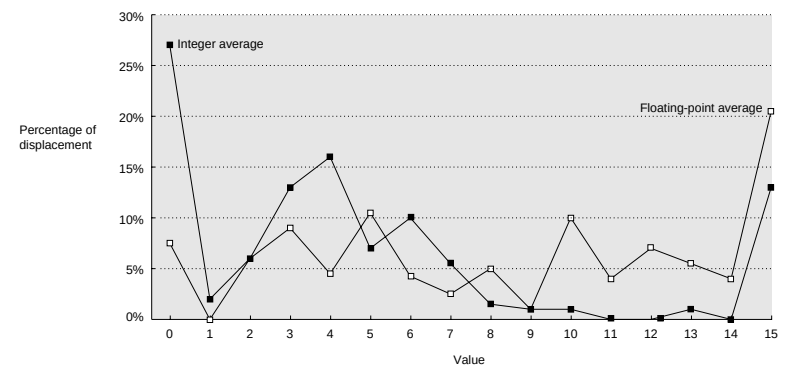
- Register direct $R3$ $R6 = R5 + R3$
- Immediate (literal) $\#25$ $R6 = R5 + 25$
- Direct (absolute) $M[10000]$ $R6 = M[10000]$
- Register indirect $M[R3]$ $R6 = M[R3]$
(a.k.a register deferred)
- Memory Indirect $M[M[R3]]$
- Displacement $M[R3 + 10000]$...
- Index $M[R3 + R4]$
- Scaled $M[R3 + R4 * d + 10000]$
- Autoincrement $M[R3++]$
- Autodecrement $M[R3--]$

Addressing Mode Utilization



Conclusion?

Displacement Size



● Conclusions?

Choice 3: Which Operations?

- arithmetic
 - add, subtract, multiply, divide
- logical
 - and, or, shift left, shift right
- data transfer
 - load word, store word
- control flow

Does it make sense to have more complex instructions?

-e.g., square root, mult-add, matrix multiply, cross product ...

CSE 240

Dean Tullsen

Types of branches (control flow)

- conditional branch beq r1,r2, label
- jump jump label
- procedure call call label
- procedure return return

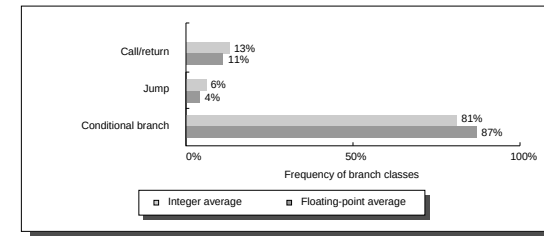


FIGURE 2.12 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches.

CSE 240

Dean Tullsen

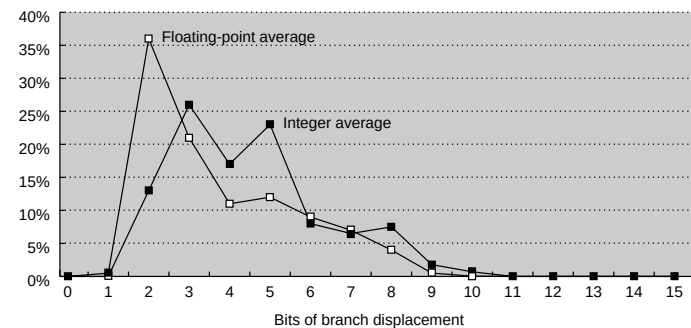
Conditional branch

- How do you specify the destination of a branch/jump?
- How do we specify the condition of the branch?

CSE 240

Dean Tullsen

Branch distance



- Conclusions?

CSE 240

Dean Tullsen

Branch condition

Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions or explicitly by compare or test instructions.

ex: sub r1, r2, r3
bz label

Condition Register

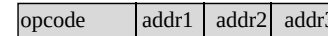
Ex: cmp r1, r2, r3
bgt r1, label

Compare and Branch

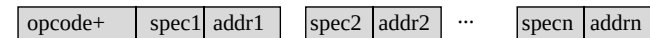
Ex: bgt r1, r2, label

Choice 4: Instruction Format

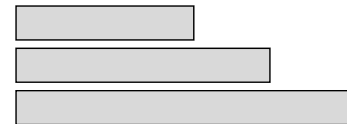
Fixed (e.g., all RISC processors -- SPARC, MIPS, Alpha)



Variable (VAX, ...)



Hybrid



- Tradeoffs?
- Conclusions?

Compiler/ISA interaction

- Compiler is primary customer of ISA
- Features the compiler doesn't use are wasted
- Register allocation is a huge contributor to performance
- Compiler-writer's job is made easier when ISA has
 - regularity
 - primitives, not solutions
 - simple trade-offs
- Summary -> simplicity over power

Our desired ISA

- Registers, Load-store
- Addressing modes
 - immediate (8-16 bits)
 - displacement (12-16 bits)
 - register deferred (register indirect)
- Support a reasonable number of operations
- Don't use condition codes
- Fixed instruction encoding/length for performance
- regularity (several general-purpose registers)

DLX instruction set architecture

- 32 32-bit general-purpose registers
 - R0 always equals zero
 - 32 or 16 FP registers
- 8-, 16-, and 32-bit integers, 32- and 64-bit fp data types
- immediate and displacement addressing modes
 - register deferred is a subset of displacement
- 32-bit fixed-length instruction encoding

CSE 240

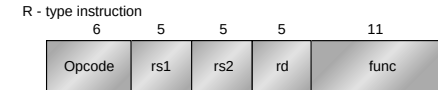
Dean Tullsen

DLX Instruction Format

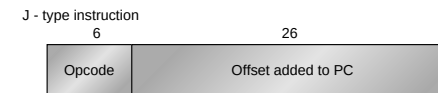


Encodes: Loads and stores of bytes, words, half words
All immediates (rd ← rs1 op immediate)

Conditional branch instructions (rs1 is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)



Register-register ALU operations: rd ← rs1 func rs2
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves



Jump and jump and link
Trap and return from exception

CSE 240

Dean Tullsen

DLX Operations

- Read on your own!
- Get comfortable with DLX instructions and formats

CSE 240

Dean Tullsen

A few sample instructions

lw R1, 1000(R2)

add R1, R2, R3

addi R1, R2, #53

JAL label

JR R3

BEQZ R5, label

CSE 240

Dean Tullsen

MIPS R2000 vs. VAX 8700

$$ET = IC * CPI * CT$$

- $IC_{MIPS} = 2 IC_{VAX}$
- $CPI_{VAX} = 6 CPI_{MIPS}$

Key Points

- Modern ISA's typically sacrifice power and flexibility for regularity and simplicity; code density for parallelism and throughput.
- instruction bits are extremely limited, particularly in a fixed-length instruction format.
- Registers are critical to performance -- we want lots of them, and few strings attached.
- Displacement addressing mode handles the vast majority of memory reference needs.