Computer Systems Architecture
CMSC 411
Unit 2 – Instruction Set Design

Alan Sussman
February 4, 2003

Instruction Set Architecture

- How instruction sets are designed
- How the design influences performance

How to design an instruction set

- How to specify data *location*
- Which instructions to include
- Which data *formats* to support
- How to *encode* instructions

Historical Perspective – Sec. 2.16

- Main points:
  - early machines had a very simple instruction set, usually stack architectures
  - in the late 1970s and early 1980s, designers tried to make the instruction set more nearly match high level languages, especially in the VAX architecture
  - in the 1980s the pendulum swung the other way: reduced instruction set computer (RISC) architectures implemented only the most frequently used instructions: Patterson and Hennessy were two of the main developers

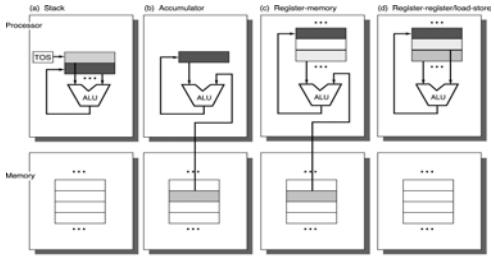
History (cont.)

- In the 1990s:
  - 64-bit addresses rather than 32 in desktop and server machines
  - instructions added to RISC to support multimedia and digital signal processing (DSP)
  - prefetch instructions to reduce penalty of data not being ready
  - floating point multiply-add instructions
- Current trends:
  - "wider" instructions (VLIW) and extending DSP processors to make compiling easier (one prime use of DSP is digital cell phones)

Types of instruction sets

- Older machines: stack or accumulator architecture
- Newer machines: load-store register architecture
  - some slightly earlier ones are register-memory

ISA classes – Figure 2.1



ISAs – Figure 2.2

C = A+B

Stack	Accumulator	Register (reg-mem)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Register architectures

- Arithmetic-logic unit (ALU) instructions can have
  - two operands, specifying two locations for the data, with the answer placed in one of those registers
  - three operands, also specifying a location for the result
- Locations can be either registers or addresses in memory
- Tradeoffs:
  - memory addresses are much longer than register addresses
  - memory is much slower than registers
  - registers are a limited resource (currently 32 or so)

Memory addressing

- Typically, memory words are **32 bits** long, divided into **4 bytes**
- Each byte has an address: the word's address, followed by the bits *00*, *01*, *10*, or *11*

Memory addressing (cont.)

- Example: Suppose that the contents of the word with address 10110 are 01101001|01111011|10000000|11110000
- **Little Endian** machines address these bytes as:

Address	Contents
1011011	01101001
1011010	01111011
1011001	10000000
1011000	11110000

Memory addressing (cont.)

- **Big Endian** machines address these bytes as:

Address	Contents
1011000	01101001
1011001	01111011
1011010	10000000
1011011	11110000

Endian-ness

- Big Endian machines: Motorola M68000, Sun Sparc, Intel IA-64
- Little Endian machines: Intel 80x86, DEC Vax, DEC/Compaq/HP Alpha
- MIPS and IBM PowerPC can do both!
- *Endian differences are a major source of bugs in transferring data between machines!*

Alignment

- If wanted a byte of data, followed by a word of data, the byte might have address *X000*, making the word's address *X001* - can shorten the number of bits in an instruction, though, if always agree to start words with addresses ending in two zeros
  - then can leave those two zeroes out of addresses involving words of memory!
- Tradeoff: waste some storage space doing this *alignment*

Alignment rules

Object addressed	Aligned at byte offsets	Misaligned at byte offsets
Byte	0,1,2,3,4,5,6,7	Never
Half word	0,2,4,6	1,3,5,7
Word	0,4	1,2,3,5,6,7
Double word	0	1,2,3,4,5,6,7

Computer Systems Architecture CMSC 411 Unit 2 – Instruction Set Design

Alan Sussman
February 6, 2003

Administrivia

- Homework #1 due today
- Homework #2 posted
  - due in 1 week – Feb. 13
- First quiz date is now Feb. 18
  - on first 2 units

Last time

- Instruction set architecture (ISA)
  - specify data location, instructions to include, data formats to support, how instructions encoded
  - 4 types of architectures
    - stack, accumulator, register-memory, load-store
  - Memory addressing
    - big-endian vs. little endian
  - Alignment
    - data item of size b is *aligned* if it is at an address that is divisible by *b*

Addressing schemes

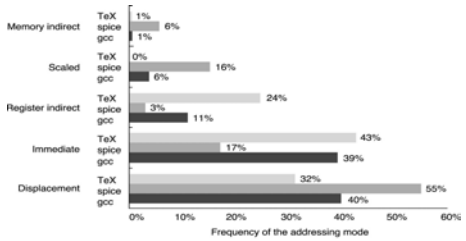
- Design addressing schemes to
  - reduce the number of bits in an instruction. examples: alignment, addressing relative to a *program counter* (PC)
  - make instructions execute quickly example: use data from registers, not from memory

Addressing modes

- A subset of Figure 2.6

Addressing modes	Example instruction	Meaning	When used
Register	Add R4,R3	R[R4]←R[R4]+ R[R3]	Value in register
Immediate	Add R4,#3	R[R4]←R[R4]+3	Constants
Displacement	Add R4,100(R1)	R[R4]←R[R4]+ Mem[100+R[R1]]	Local variables
Register indirect	Add R4,(R1)	R[R4]←R[R4]+ Mem[R[R1]]	Pointer or computed address
Indexed	Add R3,(R1+R2)	R[R3]←R[R3]+ Mem[R[R1]+R[R2]]	Array addressing
Memory indirect	Add R1,@(R3)	R[R1]←R[R1]+ Mem[Mem[R[R3]]]	Dereferencing pointer

Memory addressing mode usage



from a VAX on 3 SPEC89 programs

Addressing schemes (cont.)

- Studies of benchmarks tell computer designers which addressing schemes are most needed, and how long the addresses need to be – see pp. 97-101
- Conclusion:
  - displacement, immediate, and indirect account for more than *80%* of instructions!
  - for displacement, need 12-16 bits (Figure 2.8)
  - for immediate, need 8-16 bits (Figure 2.10)

DSP addressing schemes

- DSPs often deal with infinite, continuous data streams
  - so rely on *circular buffers*
- Implies *modulo* or *circular* addressing mode
  - start and end register with every address register, to allow autoincrement/autodecrement addressing modes to wrap around the buffer
- And for FFT (Fast Fourier Transform), *bit reverse* addressing
  - hardware reverses lower bits of address to do the shuffle FFT requires, with # bits depending on FFT algorithm step
- But the novel addressing modes are not used much
  - see Figure 2.11
  - why?

Which instructions to include?

- Again, benchmarking is essential - for 5 SPECint92 programs

Rank	**80x86 instruction**	**Integer average**
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call / return	1% / 1%
Total		96%

Operations for media/signal processing

- *SIMD* or *vector* instructions
  - to operate on multiple smaller items in one register at same time
  - lots of options about what types of instructions to support – see Figure 2.17
- DSPs also provide simple vector instructions, with *saturating arithmetic*
- DSPs also have *multiply-accumulate* (MAC) instruction
  - for vector and matrix multiply

Control Flow

- Need
  - conditional branches
  - jumps
  - procedure call / return
    sometimes including caller or callee saved registers (often done by compiler generated code, using a convention, ABI, to decide which registers caller, which callee saved)
- Addressing modes
  - *PC-relative*
    - allows using few bits, *position independence*
  - indirect – typically through a register
    - for switch stmts, virtual method calls, function pointers, dlls

How to evaluate branch conditions:

Figure 2.21

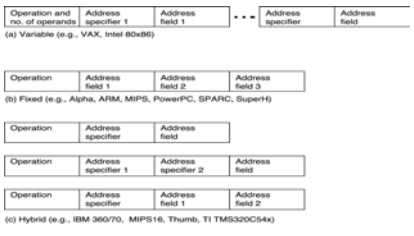
Name	Examples	How condition tested
Condition code (CC)	80x86, ARM, PowerPC, SPARC	Test special bits set by ALU ops, maybe under program control
Condition register	Alpha, MIPS	Test arbitrary register with comparison result
Compare and branch	PA-RISC, VAX	Compare is part of branch, often limited to subset of compare ops

What data formats need to be supported?

- A string of bits can mean
  - a character (1 byte, ASCII code)
  - an integer (usually 32 bits, 2's complement)
  - a floating point number (32 bits, IEEE standard format) or double precision floating point number (64 bits, IEEE standard format)
  - binary-coded decimal (4 bits for each decimal digit)
  - other less popular data types

How do we encode instructions?

- Need to have all of the necessary instructions
- Need to keep storage space small for the code



Computer Systems Architecture CMSC 411 Unit 2 – Instruction Set Design

Alan Sussman

February 11, 2003

Administrivia

- Homework #2 due Thursday
  - for problem 2.1b, compute the answer in terms of the size of a data element, *n*
- Homework #3 handed out Thursday too
  - due date not set yet
- First quiz next Tuesday, on first 2 units

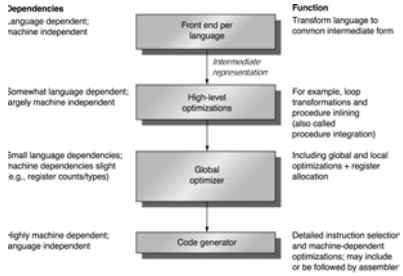
Last time

- Addressing schemes
  - register, immediate, displacement (includes register indirect, absolute), indexed, memory indirect
  - additional DSP modes – modulo, bit reverse
    - not used much, since compilers don't use them
- Instructions
  - top 10 instructions account for 96% of instructions executed on SPECint92 benchmarks
  - DSP/multimedia inst. include SIMD inst (sometimes with saturating arithmetic) and MAC
  - control flow inst. – cond. branches, jumps, procedure call/return – with PC-relative addressing or indirect
  - evaluate branch conditions via condition codes, condition register, or compare & branch

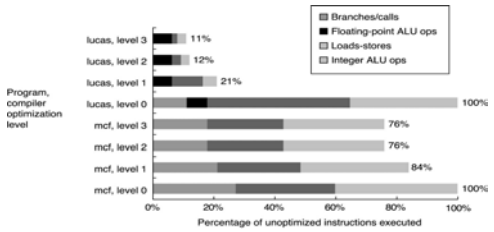
Interaction with the compiler

- Compilers need to
  - allocate registers (multiple roles for each)
  - perform optimization (See Figure 2.25 for more detail)
  - manage data structures efficiently:
    - a stack for local variables and change of state
    - global data area
    - heap

Compiler phases/passes – Fig. 2.24



Effects of compiler optimization



SPEC2000 on Alpha – Fig. 2.26

Architect can help compiler writer

- Provide regularity
  - operations, data types, addressing modes should be *orthogonal*
- Provide primitives, not solutions
  - no "special features"
- Simplify tradeoffs among alternatives
  - to determine costs of different code sequences
- Allow compiler to bind quantities known at compile-time
  - if a value is known then, let the compiler tell the hardware (in instruction, register, etc.)

MIPS architecture design criteria

- For desktop applications - see Section 2.12
  - General purpose registers with a load-store architecture
  - Addressing modes:
    - displacement (offset 12-16 bits)
    - immediate (8-16 bits)
    - register indirect
  - Data types
    - 8-, 16-, 32-, 64-bit integers
    - 64-bit IEEE 754 floating point numbers
  - Simple instructions
    - load, store, add, subtract, move register-register, shift

MIPS architecture (cont.)

- Control flow
  - Compare =,!=,< , branch (PC relative >= 8 bits), jump, call, return
- Fixed instruction encoding for performance (desktop machines), variable encoding for code size reduction (embedded processors)
- At least 16 general-purpose registers
  - all addressing modes apply to all data transfer instructions
- Often separate floating point registers
- More on instruction set in Appendix C, online

MIPS64 registers

- 32 64-bit general purpose (integer) registers (GPRs)
  - R0, R1, …, R31
- 32 floating point registers (FPRs)
  - F0, F1, …, F31
  - hold 32 single-precision (32-bit) or 32 double-precision (64-bit) values
  - both single- and double-precision floating point ops
  - and instructions to operate on 2 single-precision operands in one FPR
- R0 always contains 0
- Instructions for moving between FPR and GPR

MIPS64 addressing modes

- Immediate and displacement
  - get register indirect by setting displacement field to 0
  - get absolute addressing by using register 0 (with value 0) as base register for immediate mode
- Memory byte addressable with 64-bit addresses
  - mode bit sets Big or Little Endian
  - all memory references between memory and registers through loads and stores
  - GPR memory accesses byte, half word, word, double word
  - FPR load/store with single- or double-precision
  - all memory accesses must be aligned

MIPS64 instruction formats

- Addressing mode encoded in opcode (1 bit)
- All instructions 32 bits, 6-bit primary opcode

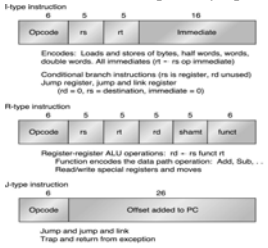


Figure 2.27

MIPS64 Loads and Stores

- Can use either memory addressing mode
  - GPRs – byte, half word, word, double word
  - FPRs – single- and double-precision

ALU instructions

- All register-register instructions
- Example: A = A + B, A stored at address 2000, B stored at address 1500, A and B integers (Addresses in octal)

```
LD    R1, 2000(R0)
LD    R2, 1500(R0)
DADD  R1, R1, R2
SD    R1, 2000(R0)
```

- How would the code change if A and B were floats?

MIPS64 Floating point operations

- Manipulate FPRs and indicate single- or double-precision
- Instructions for:
  - copying single- or double-precision FPR to another FPR
  - moving data between FPR and GPR
  - conversions between integer/floating point
  - arithmetic: add, subtract, multiply, divide
    - single- and double-precision
  - comparisons – sets bit in FP status register that can be tested with branch true/false instructions
  - *paired single* operations – perform 2 32-bit FP ops on each half of a 64-bit FPR

MIPS64 Control: Jumps and Branches

- Example: Suppose integer array *a* is stored beginning at location 3000 and i is stored at 2500

```
i=0;                        DADDI  R3,R0, #1
while (a[i] < 0){           DSLL   R3,R3, #63(decimal)
  i++;                      DADD   R1,R0,R0     i accumulated here
}                           DADD   R4,R0,R0     4*i accumulated here
                          jmppt:  LD    R2, 3000(R4)
                            AND    R2,R3,R2
                            BEQZ   R2, fin
                            DADDI  R4,R4, #8
                            DADDI  R1,R1, #1
                            J      jmppt
                          fin:    SD    R1, 2500(R0)
```


Important note

- There is **not** enough information in Chapter 2 to enable you to use every MIPS64 instruction correctly
- For example, H&P never explain the format of the floating point status register
- Don't get frustrated by this; just make a good-faith effort to write legal code, and we'll fill in more details as we need them.
- More on instruction set in Appendix C, online
- Can also look at embedded (media) processor example in Section 2.13

Pitfalls

- **An instruction set closer to a high level language is more efficient**
  - more instructions make the op-codes longer - wastes space
  - most instructions are seldom used
  - instructions that are too specialized often do more work than usually necessary, slowing execution down
- RISC usually beats these designs

Pitfalls (cont.)

- **Not considering the compiler**
  - compiler strategies can make a huge difference in code size (probably a lot more than hardware innovations to save space)

Fallacies

- **Design a computer for a typical program**
  - there is no such thing as a typical program!
- **Design a perfect architecture**
  - there is no such thing as a perfect architecture: there are always tradeoffs

Fallacies (cont.)

- **Architectures with flaws cannot be successful**
  - counter-example is the 80x86
  - ISA is a mess, but obviously commercially successful – why?
    - since it was in the original IBM PC, binary compatibility very valuable
    - Moore's law provide enough resources to translate internally to RISC ISA, and execute that way
    - High volume of PC microprocessors allows Intel to pay for increased design costs of hardware translation