# ECE 486/586

# Computer Architecture

# Lecture # 7

Spring 2015

Portland State University

## Lecture Topics

- Instruction Set Principles
  - Instruction Encoding
  - Role of Compilers
  - The MIPS Architecture

Reference:

- Appendix A: Sections A.6, A.7, A.8 and A.9

## Encoding Instructions

- OpCode – Operation Code
  - The instruction (e.g., “add”, “load”)
  - Possible variants (e.g., “load byte”, “load word”…)
- Source and Destination
  - Register or memory address
- Addressing Modes
  - Impacts code size
  - Two options:
    - Encode as part of opcode (common in load-store architectures which use a few number of addressing modes)
    - Address specifier for each operand (common in architectures which support may different addressing modes)

## Encoding Instructions

- Tradeoff between size of program versus ease of decoding
- Must balance the following competing requirements:
  - Support as many registers and addressing modes as possible
  - Impact of size of the register and addressing mode fields on the average instruction size
  - Desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation

# Encoding Instructions

| Operation and no. of operands | Address specifier 1 | Address field 1 | • • • | Address specifier *n* | Address field *n* |
|---|---|---|---|---|---|

(a) Variable (e.g., Intel 80x86, VAX)

| Operation | Address field 1 | Address field 2 | Address field 3 |
|---|---|---|---|

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

| Operation | Address specifier | Address field |
|---|---|---|

| Operation | Address specifier 1 | Address specifier 2 | Address field |
|---|---|---|---|

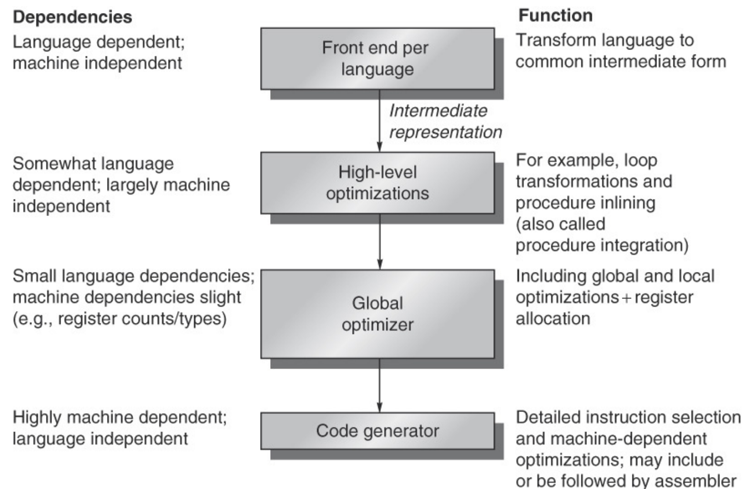
| Operation | Address specifier | Address field 1 | Address field 2 |
|---|---|---|---|

(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

# Fixed vs. Variable Length Encoding

- Fixed Length
  - Simple, easily decoded
  - Larger code size
- Variable Length
  - More complex, harder to decode
  - More compact, efficient use of memory
    - Fewer memory references
    - Advantage possibly mitigated by RISC use of cache
  - Complex pipeline: instructions vary greatly in both size and amount of work to be performed

# Structure of Compilers




# Compiler Optimizations

- High-level optimizations done on source
  - Procedure in-lining
    - Replace “expensive” procedure calls with in-line code
- Local optimizations within a “basic block”
  - Common sub-expression elimination (CSE)
    - Don’t re-evaluate the same sub-expression; save result and reuse
  - Constant propagation
    - Replace all instances of variable containing a constant with the constant
    - Reduces memory accesses (immediate mode)

## Compiler Optimizations (cont.)

- Global optimizations done across branches
  - Copy propagation
    - Replace all instances of variable assignments A = X with X
  - Code motion
    - Remove invariant code from loop
- Register Allocation
  - Allocate registers to expression evaluation, parameter passing, variables etc.

## Compiler Optimizations (cont.)

- Processor dependent optimizations
  - Strength reduction
    - Replace/choose instructions with less “expensive” alternatives
    - Example: * 2 and / 2 by left shift or right shift
  - Pipeline scheduling
    - Re-order instructions to improve pipeline performance

## How the Computer Architect can help the Compiler Writer

- Provide regularity
  - Make operations, data types, addressing modes *orthogonal*
  - Example: For every operation to which one addressing mode can be applied, all addressing modes are applicable
- Provide primitives, not solutions
  - Special instructions that “match” a high-level language construct are often unusable
- Simplify trade-offs among alternatives
  - Make it easy for the compiler writer to determine the most effective implementation for a particular instruction sequence

## RISC vs CISC Debate

- CISC (VAX)
  - Attempts to create instructions to support high-level languages
    - Procedure calls
    - String processing
    - Loops, array accesses
  - Complex architecture, harder to pipeline
- RISC (MIPS, ARM)
  - Use simpler architecture
    - Simpler implementation → faster clock cycle
    - Regularity of instruction format → Easier to pipeline
    - Make the common case fast, combination of instructions for less frequent cases

# Introduction to MIPS

- Overview of MIPS instruction set architecture (ISA)
  - Assembly language
  - Machine code
- Why study MIPS?
  - Easy architecture to understand
  - Understand and create assembly language examples for rest of course
  - Provides a framework to understand ISA tradeoffs
  - No need to become a MIPS assembly language expert

# The MIPS Architecture

- Use general-purpose registers with a load-store architecture
- Support most common addressing modes
  - *Register, immediate, displacement*
- Support 8-, 16-, 32- and 64-bit integers and 32- and 64-bit IEEE 754 floating point numbers
- Focus on most commonly executed instructions
  - *Load, store, add, subtract, move, shift*
- Use small number of control instructions
  - *compare {equal, not equal}*
  - *branch PC-relative*
  - *jump, jump and link (JAL), jump register (JR)*
- Use fixed length instruction encoding

# MIPS Registers

- Arithmetic instruction operands must be registers
  - 32 integer general-purpose registers: R0, R1, …., R31
    - Value of R0 is always 0
  - 32 floating point registers: F0, F1, …., F31
- Compiler associates variables with registers
- What about a program with lots of variables?

# Memory Organization

- Viewed as a large, single-dimensional array, with addresses
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory

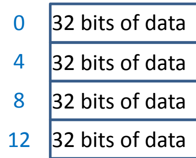
| Address | Contents |
|---|---|
| 0 | 8 bits of data |
| 1 | 8 bits of data |
| 2 | 8 bits of data |
| 3 | 8 bits of data |

….

….

# Memory Organization

- Data items in typical programs need more than one byte => support needed for larger “words”
- For MIPS, a word is 32 bits or 4 bytes



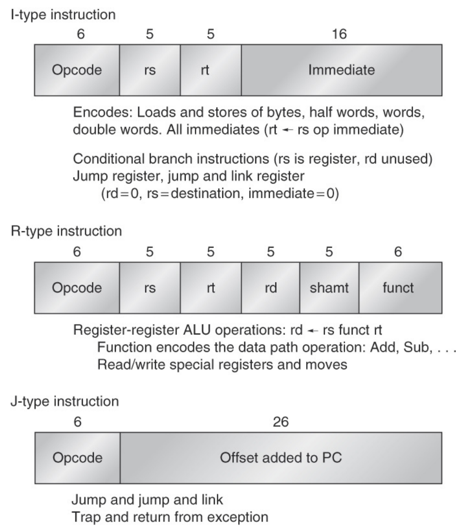

....

- $2^{32}$ bytes with byte addresses from 0 to $2^{32}$-1
- $2^{30}$ words with byte addresses 0, 4, 8, …. , $2^{32}$-4
- Words are aligned
  - What are the least 2 significant bits of a word address?

# MIPS Instruction Encoding

- All instructions are 32-bits (fixed size) with a 6-bit opcode
  - Easy to decode and pipeline
- Basic instruction types
  - Arithmetic/Logic
  - Loads/Stores
  - Control
- Three instruction formats
  - I-type
  - R-type
  - J-type

# MIPS Instruction Encoding (cont.)




# ALU Instructions

- All ALU instructions have three operands
- Operand order is fixed: *destination*, *source1*, *source2*

```
a = b + c;    =>   add r17, r18, r19
```

- Keep instruction results in registers if possible

```
a = b + c;    =>   add r17, r18, r19
f = a + e;         add r17, r17, r20
```

- Machine code representation of first `add` instruction:

| R | op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| | 000000 | 10010 | 10011 | 10001 | 00000 | 100000 |

- opcode = 0; funct determines specific ALU operation

## ALU Instructions (cont.)

- Immediate operands

```
a = b + 16;     =>   addi r17, r18, 16
```

- Immediate mode form of ALU operations
  - Opcode encodes specific operation (and indicates immediate)
  - 16-bit immediate value in low-order 16-bits

| I | op | rs | rt | Immediate |
|---|---|---|---|---|
| | op | 10010 | 10001 | 0000000000010000 |

## Load/Store Instructions

- Only instructions which access memory
- Displacement mode addressing

```
count[4] = x + count[2]; =>   lw r8, 8(r20)
                              add r8, r19, r8
                              sw r8, 16(r20)
```

  - r20: base address of “count” array
  - r19: value of “x”
- Machine code representation of “lw” instruction

| I | op | rs | rt | Immediate |
|---|---|---|---|---|
| | op | 10100 | 01000 | 0000000000001000 |

## Control Flow Instructions

- Conditional Branches
- Jumps
  - Unconditional
- Procedure calls
- Procedure returns