Previous lecture recap

- Metrics of computer architecture
- Fundamental ways of improving performance: parallelism, locality, focus on the common case
- Amdahl's Law: speedup proportional only to the affected fraction of the original execution time
- CPU Performance equation: IC * CPI * Clock time
  - Must improve some combination of the above to improve perf

Reminder: tutorials start next week!

ISA: The Hardware – Software Interface

- Instruction Set Architecture (ISA) is where software meets hardware
  - Understanding of ISA design is therefore important

- Instruction Set Components
  - Operands: int32, uint32, int16, uint16, int8, uint8, float32, float64
  - Addressing modes: how do we access data (in regs, memory, etc)
  - Operations: four major types
    - Operator functions (add, shift, xor, mul, etc)
    - Data movement (load-word, store-byte, etc)
    - Control transfer (branch, jump, call, return, etc)
    - Privileged, and miscellaneous instructions (not part of the application)

- Good understanding of compiler translation is essential

ISA Design Considerations

- Simple target for compilers
- Support for OS and programming language features
- Support for important data types (floating-point, vectors)
- Code size
- Impact on execution efficiency (especially with pipelining)
- Backwards compatibility with legacy processors
- Provision for extensions

CISC vs RISC

- CISC
  - Assembly programming → HLL features as instructions
  - Small # registers (but memory “fast”) → in-memory operands
  - Code size must be small (transistors scarce) → variable length instructions
  - Backward compatibility → complexity grows over time

- RISC
  - Compilers → Simple instructions
  - Large # registers, memory much slower than processor → load-store architecture
  - Simple and fast decoding → fixed length, fixed format

Instruction Classes

- Instructions that operate on data
  - Arithmetic & logic operations
  - Execution template: fetch operands, perform op, store result
- Instructions that move data
  - Move data between registers, memory, and I/O devices
- Instructions that change control flow
  - Re-direct control flow away from the next instruction
  - May be conditional or unconditional (including exceptions!)

Operators and their Instructions

- Integer Arithmetic

+	add
-	sub
*	mul
/	div
%	rem

- Relational

<	slt, sltu
<=	sle, sleu
>	sgt, sgtu
>=	sge, sgeu
==	seq
!=	sne

C operator	Comparison	Reverse	Branch
==	seq	0	bnez
!=	seq	0	beqz
<	slt, sltu	0	bnez
>=	slt, sltu	0	beqz
>	slt, sltu	1	bnez
<=	slt, sltu	1	beqz

Operators continued...

- Bit-wise logic
  - | or
  - & and
  - ^ xor
  - ~ not
- Boolean
  - || (src1 != 0 or src2 != 0)
  - && (src1 != 0 and src2 != 0)
- Shifts
  - >> (signed) shift-right-arithmetic
  - >> (unsigned) shift-right-logical
  - << shift-left-logical

Operand Types

- Usually based on scalar types in C

Type modifier	C type declarator	Machine type
unsigned	int, long	uint32
unsigned	short	uint16
unsigned	char	uint8
unsigned	long long	uint64
signed	int	int32
signed	short	int16
signed	char	int8
signed	long long	int64
	float	float32
	double	float64
	&<type_specifier>	uint32

This assumes a 32-bit machine!

- C defines integer promotion for expression evaluation
  - **int16** + **int32** will be performed at 32-bit precision
    - First operand must be sign-extended to 32 bits
  - Similarly, **uint8** + **int16** will be performed at 16-bit precision
    - First operand must be zero-extended to 16-bit precision
    - Result (signed or unsigned) determined by the operand's type in the source code

Instruction Operands - Registers

- How many register-based operands should be specified?

```
3:   R1 = R2 + R3
2:   R1 = R1 + R2
1:   +R1
```

- 32-bit RISC architectures normally specify 3 registers for dyadic operations and 2 registers for monadic operations
- Compact 16-bit embedded architectures often specify respectively 2 and 1 register in these cases
  - Reduces cost through fewer ports in the register file, less wire routing, etc.
  - "Destructive" ops → Requires extra register copying to preserve original values
  - E.g.

```
load   r1, [address]
copy   r2, r1
add    r1, r3
sub    r4, r2     # this is simply a re-use of r1, but the value of r1 had to be copied
                    into r2
```

- Accumulator architectures: now dead, but concept still widely used in Digital Signal Processors (DSP).
  - E.g.

```
load [address1]
add   23
store [address2]
```

Register (accumulator) is implicit

Instruction Operands - Literals

- Constant operands
  - E.g. add r1, r2, 45

- Jump or branch targets
  - Relative:
    - Normally used for if-then-else and loop constructs within a single function
    - Distances normally short – can be specified as 16-bit signed & scaled offset
    - Permits “position independent code” (PIC)
  - Absolute
    - Normally used for function call and return
    - But not all function addresses are compile-time constants, so jump to *contents* of register is also necessary for flexibility

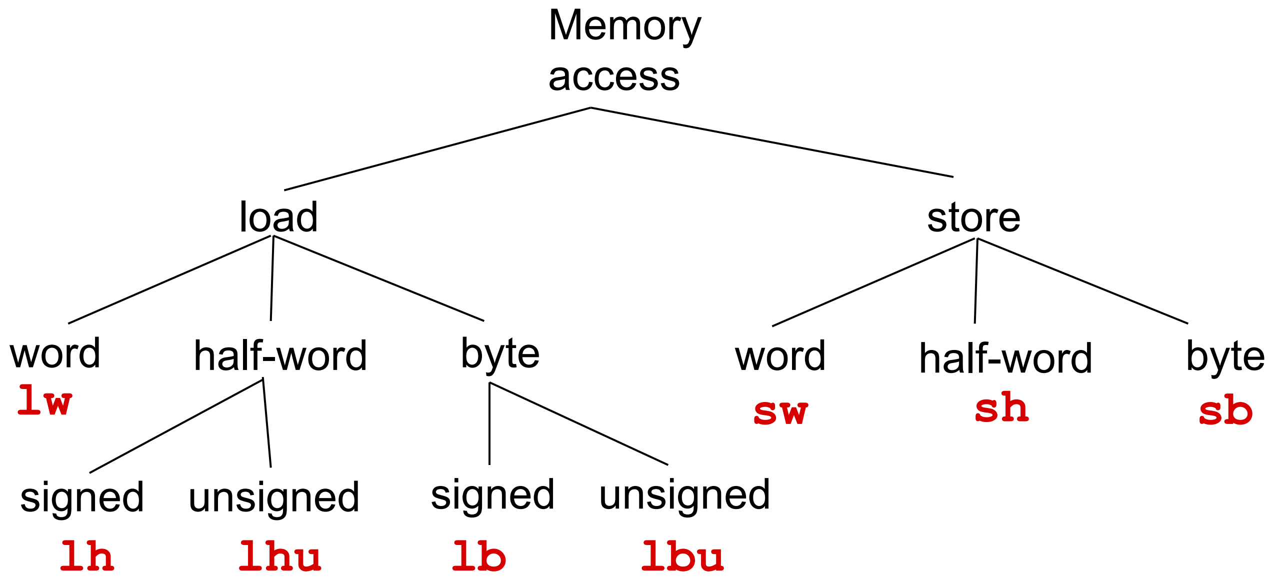
- Load/Store addresses
  - Relative
  - Absolute

How big do literals have to be?

- Addresses
  - Fixed & machine-specific: typically 32 or 64 bits
- Arithmetic operands
  - Small numbers, typically representable in 5 – 10 bits
- Literals are often used repeatedly at different locations
  - Place as read-only data in the code and access relative to program counter register (e.g. MIPS16, ARM-thumb)
- Branch offsets
  - 10 bits catches most branch distances
- 32-bit RISC architectures provide 16-bit literals
- 16-bit instructions must cope with 5 – 10 bits
  - May extend literal using an instruction prefix
  - E.g. ARM Thumb bx instruction → can branch conditionally to a target specified in a register, thus offering a 32-bit displacement

Memory Access Operations

- Memory operations are governed by:
  - Direction of movement (load or store)
  - Size of data objects (word, half-word, byte)
  - Extension semantics for load data (zero-ext, sign-ext)



Memory Addressing Modes: Displacement

Displacement addressing is the most common memory addressing mode

- Register + offset
  - Generic form for accessing via pointers
  - Multi-dimensional arrays require address calculations
- Stack pointer and Frame pointer relative
  - 5 to 10 bits of offset is sufficient in most cases
- PC relative addresses
  - Used to modify control flow (e.g., upon a branch)
  - Also to access literals (see earlier slide)

Other Memory Addressing Modes

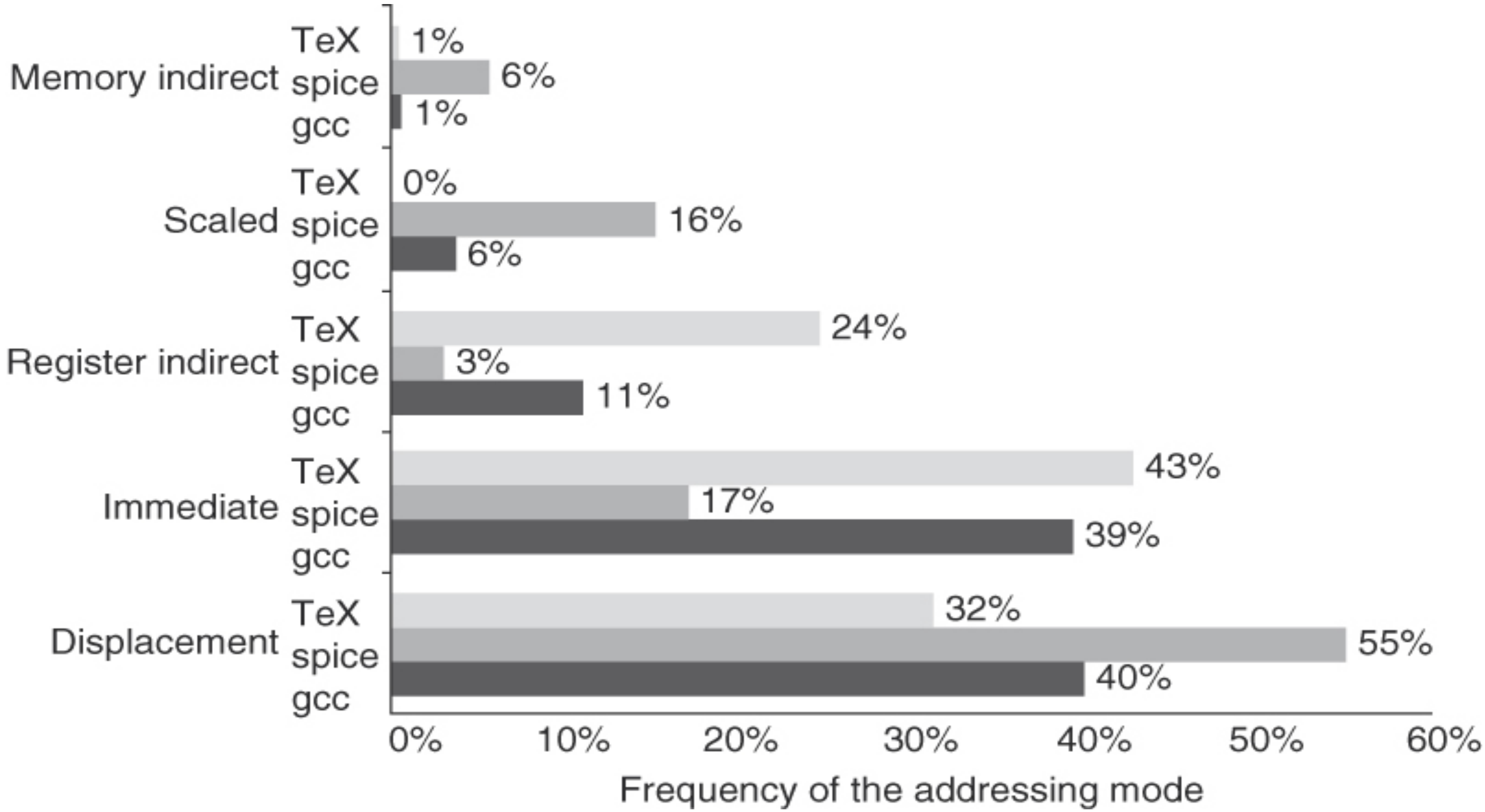
- Direct or absolute: useful for accessing constants and static data
- Auto-increment/decrement: useful for iterating over arrays or for stack push/pop operations
- Scaled: speeds up random array accesses

  e.g., `R7 = R5 + Mem[R1 + R2 * d]`

  where *d* is determined by the size of the data item being accessed (byte, hw, word, long)
- Memory indirect: in-memory pointer dereference

  e.g., `R3 = Mem[Mem[R1]]`

Memory Addressing Mode Frequency

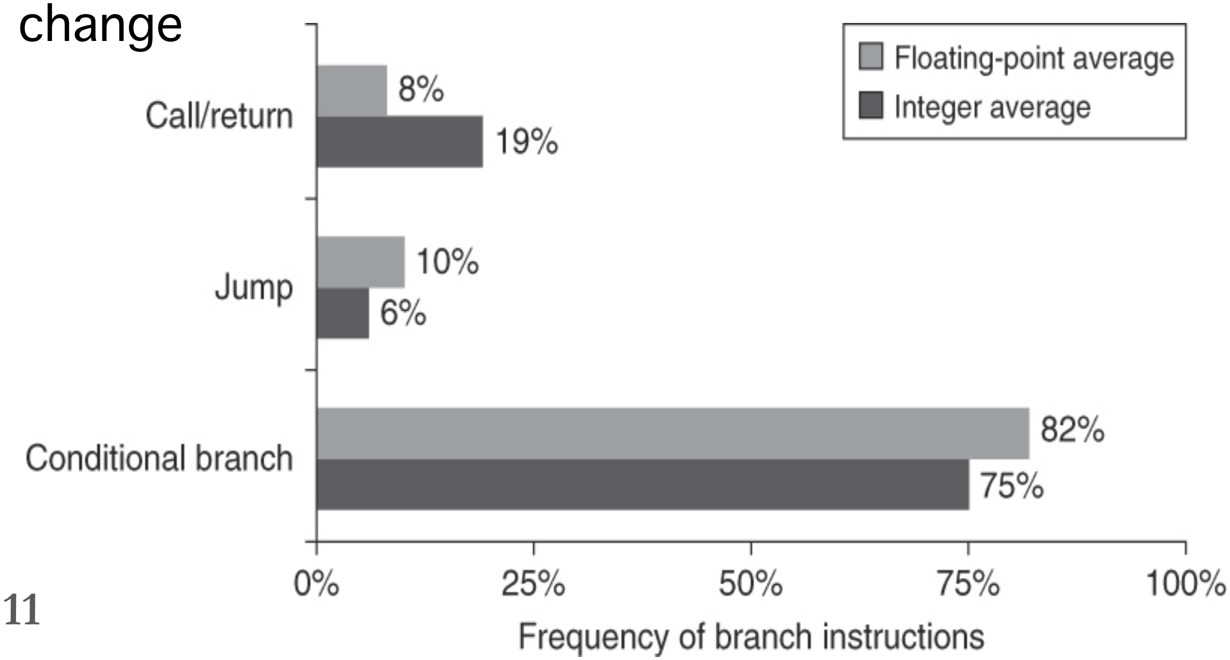


H&P 5/e Fig. A.7

Few addressing modes account for most memory accesses

Instructions for Altering Control Flow

- Conditional (branches)
- (unconditional) Jumps
- Function calls and returns
- Exceptions & interrupts
  - Traps (instructions) vs events
  - Trigger a mode change



H&P 5/e Fig. A.11

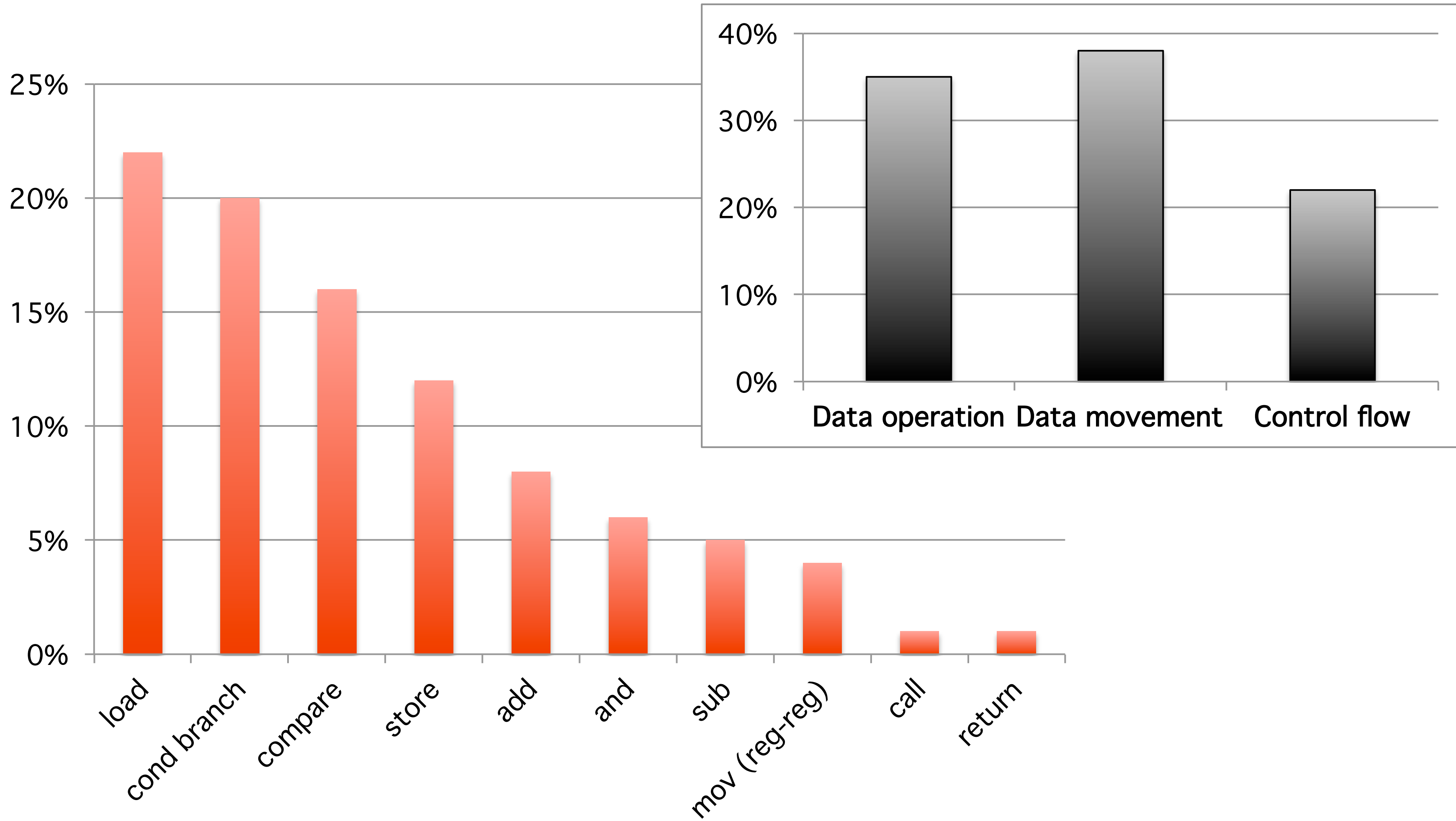
Conditional Instruction Formats

- Condition code based (e.g., x86)
  - sub $1, $2
  - Sets Z, N, C, V flags
  - Branch selects condition
    - ble : N or Z
  - (+) Condition set for free (“side-effect” of instruction execution)
  - (-) Volatile state (next instruction may overwrite flags)

- Condition register based
  - slte $1, $2, $3
  - bnez $1 (or beqz $1)
  - (+) Simple and reduces number of opcodes
  - (-) Uses up a register

- Compare and branch
  - combt lte $1, $2
  - (+) One instruction per branch
  - (-) “Complex” instruction

Instruction Frequency by Type



Data from H&P 5/e Fig. A.13

Encoding the Instruction Set

- How many bits per instruction?
  - Fixed-length 32-bit RISC encoding
  - Variable-length encoding (e.g. Intel x86)
  - Compact 16-bit RISC encodings
    - ARM Thumb
    - MIPS16
- Formats define instruction groups with a common set of operands
- Orthogonal ISA: addressing modes are independent of the instruction type (i.e., all insts can use all addressing modes)
  - Great conceptually and for compilation
  - E.g., VAX-11: 256 opcodes * 13 addressing modes (mode encoded with each operand)

MIPS 32-bit Instruction Formats

- R-type (register to register)
  - three register operands
  - most arithmetic, logical and shift instructions
- I-type (register with immediate)
  - instructions which use two registers and a constant
  - arithmetic/logical with immediate operand
  - load and store
  - branch instructions with relative branch distance
- J-type (jump)
  - jump instructions with a 26 bit address

MIPS R-type instruction format

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	reg rs	reg rt	reg rd	shamt	funct

```
add    $1, $2, $3
```

special	$2	$3	$1		add

```
sll    $4, $5, 16
```

special		$5	$4	16	sll

MIPS I-type instruction format

6 bits	5 bits	5 bits	16 bits
opcode	reg rs	reg rt	immediate value/addr

`lw $1, offset($2)`	lw	$2	$1	address offset
`beq $4, $5, .L001`	beq	$4	$5	(PC - .L001) >> 2
`addi $1, $2, -10`	addi	$2	$1	0xfff6

MIPS J-type instruction format

6 bits	26 bits
opcode	address

`call func`

`jal`	`absolute func address >> 2`

ISA Guidelines

- Regularity: operations, data types, addressing modes, and registers should be independent (orthogonal)
- Primitives, not solutions: do not attempt to match HLL constructs through the instruction set
- Simplify tradeoffs: make it easy for compiler to make choices based on estimated performance