Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 6:   Sign Extension, Registers vs. Memory, Logical Operations, and Instructions.

Taylor Johnson

Important Concepts from Previous Lectures

- Signed Numbers, Hexadecimal, Instructions, and Endianess

- Instruction Set
  - MIPS
  - ARM

- Arithmetic Operations

- Some Processor Components
  - Register Operands
  - Memory Operands

Announcements and Outline

- Homework 2 on course website
  - Read chapter 2 (ARM version on Blackboard site)

- Review from last time / Chapter 2
  - Instruction Set, Arithmetic Operations
- Sign Extension, Registers vs. Memory, Logical Operations, and Instructions

Review: Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **Range: 0 to $+2^n - 1$**

- **Example**

  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1{\times}2^3 + 0{\times}2^2 + 1{\times}2^1 + 1{\times}2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Review: 2s-Complement Signed Integers

- ## Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1{\times}2^{31} + 1{\times}2^{30} + \ldots + 1{\times}2^2 + 0{\times}2^1 + 0{\times}2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- ## Using 32 bits
  - $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$

Review: Two's Complement Signed Negation

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010$_2$
  - −2 = 1111 1111 … 1101$_2$ + 1

    = 1111 1111 … 1110$_2$

Review: Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
  - Will discuss a few in this course, primarily will focus on ARM for assignments
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Review: MIPS and ARM Instruction Sets

- MIPS
  - Used as examples throughout the book
  - Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Large share of embedded core market
    - Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - Typical of many modern ISAs
    - See MIPS Reference Data tear-out card, and Appendixes B and E
- ARM
  - Commercially much more successful (nearly every phone)
  - Similar to MIPS
  - ARM version of chapters on Blackboard
  - Use this for programming assignments

Review: Arithmetic Operations

- Add and subtract, three **_operands_**
  - **_Operand:_** *quantity on which an operation is performed*
  - Two sources and one destination

```
add a, b, c  # a updated to b + c
```

- All arithmetic operations have this form

- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Review: Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

- Compiled ARM code:

```
add r0, g, h    # temp r0 = g + h
add r1, i, j    # temp r1 = i + j
sub f, r0, r1   # f = t0 - t1
```
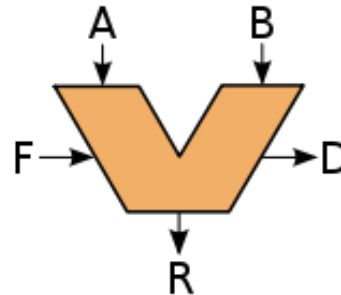
- Notice: registers "=" variables

Review: Some Processor Components

CPU

Register File

- Program Counter (PC)
- Instruction Register (IR)
- General Purpose Registers
  - Word size
  - Typically 16-32 of these
  - PC sometimes one of these
- Floating Point Registers

Arithmetic logic unit (ALU)

Floating Point Unit (FPU)

Review: Register Operands

- Arithmetic instructions use register operands

- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"

- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Review: ARM 7 Registers

- 16 32-bit general purpose registers
- 32 32-bit floating-point registers (not available on every device)

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM's general registers.

Review: ARM 7 Registers

- The Vx registers hold data needed by procedures (functions)
- They should be stored in memory when calling another procedure
- They should be restored from memory when returning from another procedure

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM's general registers.

Review: ARM 7 Registers

- The Ax registers are used for passing parameters to procedures

- Four dedicated registers have special roles: IP, SP, LR, PC.
  - We will see more details on these registers are later

- Who ensures that these registers are used as specified here?
  - **You!!! (The programmer)**

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Version 7 ARM's general registers.

Review: ARM: Load/Store Architecture

- ARM is a load/store architecture
- This means that memory can only be accessed by load and store instructions
- All arguments for arithmetic and logical instructions must either:
  - Come from registers
  - Be constants specified within the instruction
    - (more examples of that later)
- This may not seem like a big deal to you, as you have not experienced the alternative
  - However, it makes life much easier
  - This is one reason why we chose ARM 7 for this course

Review: Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS/ARM are Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Review: Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - `f, …, j` in:
    - `$s0, …, $s4 (MIPS)`
    - `r0, …, r4 (ARM)`

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

- Compiled ARM code:

```
add r0, r1, r2
add r1, r3, r4 // overwrite r1
sub r0, r0, r1
```

- Note: syntax and semantics (meaning) differences

Byte Ordering - Endianness

- How do we store an integer in memory?

- Simple answer: in binary

- Actual answer: yes, in binary, but this does not fully specify how we store the number

- Unfortunately, we have two choices

- Common architectures may follow either choice, and mess ensues, unless we are aware of this issue and we deal with it explicitly

- This is the problem of **endianness**

Endianness

- Little-endian: increasing numeric significance with increasing memory addresses
- Big-endian: decreasing numeric significance with increasing memory addresses
- Little-Endian Examples
  - x86, x86-64, 8051, DEC Alpha, Atmel AVR
- Big-Endian Examples
  - Motorola 6800 and 68k series, Xilinx Microblaze, IBM POWER, and System/360
- Bi-Endianness
  - Ability for computer to operate using either
  - SPARC
  - ARM architecture: little-endian before version 3, now bi-endian

Endianness Example

Big-endian
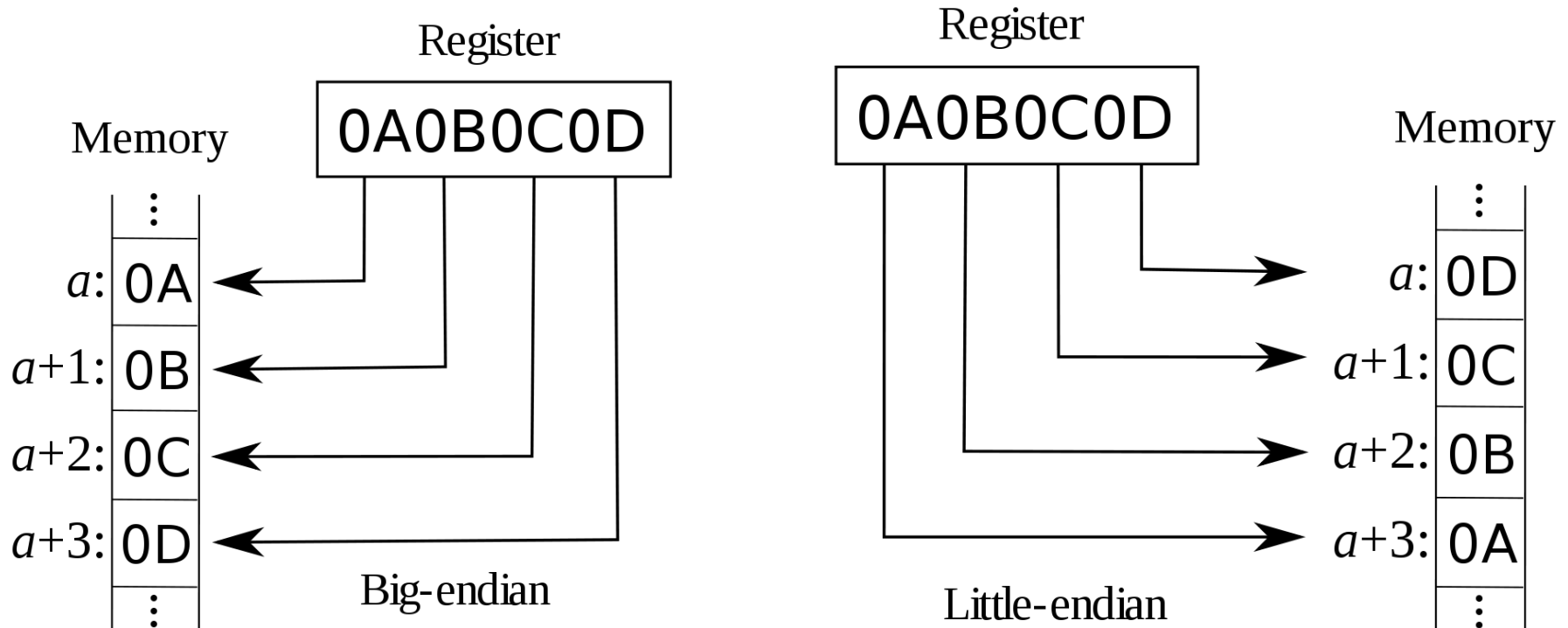
Little-endian

Byte Ordering Visualization
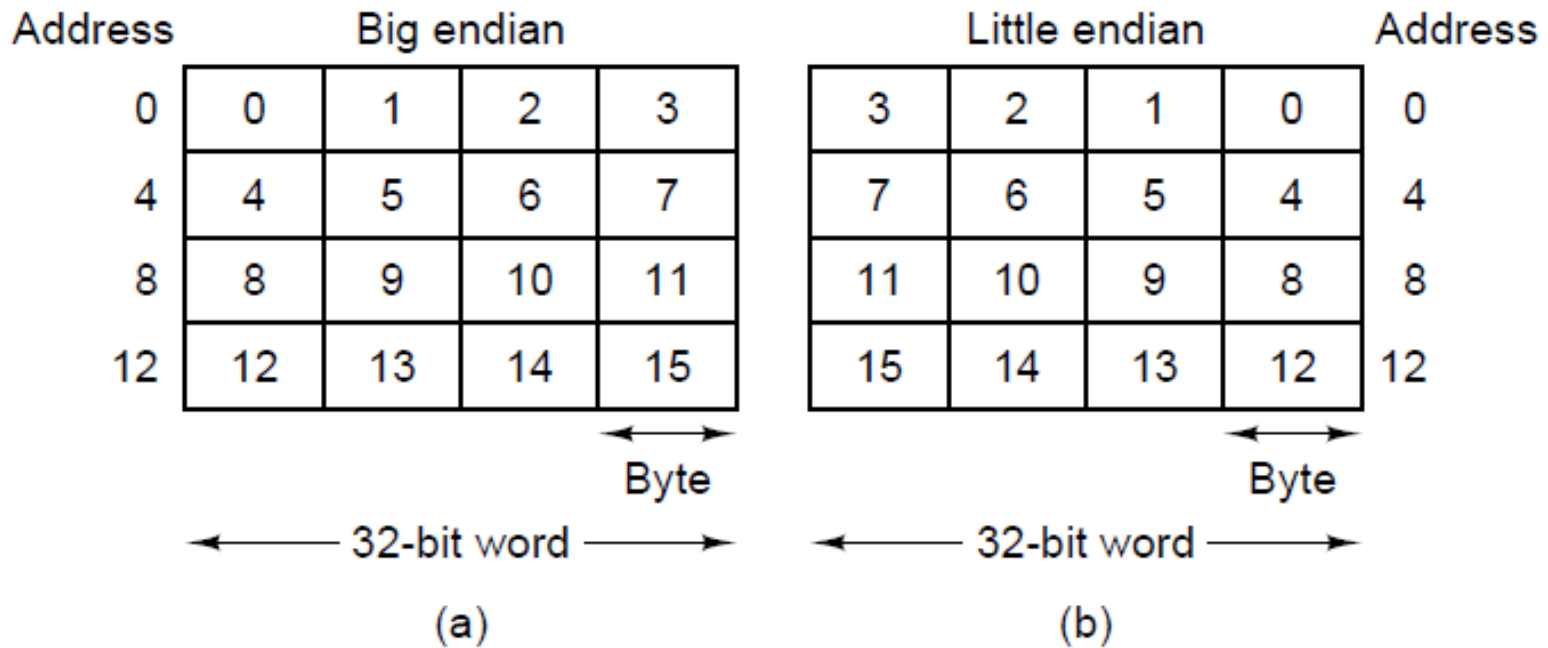
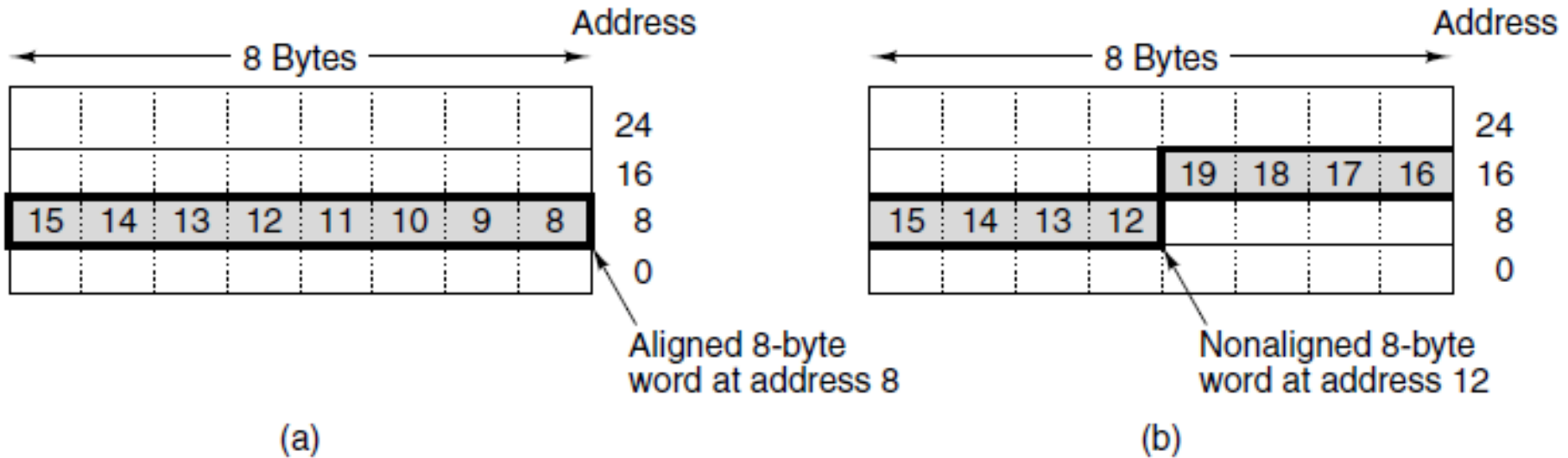(a) Big endian memory. (b) Little endian memory.

Main difference: ordering of bytes in a word

- Left-to-right in big endian.

- Right-to-left in little-endian.

Memory: Words and Alignment

- Bytes are grouped into **words**
- Depending on the machine, a word can be:
  - 32 bits (4 bytes) , or
  - 64 bits (8 bytes), or … (16-bits, 128 bits, etc.)
- Oftentimes it is required that words are aligned
- This means that:
  - 4-byte words can only begin at memory addresses that are multiples of 4: 0, 4, 8, 12, 16…
  - 8-byte words can only begin at memory addresses that are multiples of 8: 0, 8, 16, 24, 32, …

Memory Models

An 8-byte word in a little-endian memory.
(a) Aligned. (b) Not aligned. Some machines
require that words in memory be aligned.

Memory Cells and Addresses

- Memory cell: a piece of memory that contains a specific number of bits
  - How many bits depends on the architecture
  - In modern architectures, it is almost universal that a cell contains 8 bits (1 byte), and that will be also our convention in this course
- Memory address: a number specifying a location of a memory cell containing data
  - Essentially, a number specifying the location of a byte of memory

Memory Cells and Addresses

- The number of unique memory addresses depends on the size of the memory and the size of each cell

- For example, suppose we have a 96-bit memory.

- If each cell is 8 bits, we have ??? addresses?

- If each cell is 12 bits, we have ??? addresses?

- If each cell is 16 bits, we have ??? addresses?

Memory Cells and Addresses

- The number of unique memory addresses depends on the size of the memory and the size of each cell

- For example, suppose we have a 96-bit memory.

- If each cell is 8 bits, we have 12 addresses?

- If each cell is 12 bits, we have 8 addresses?

- If each cell is 16 bits, we have 6 addresses?

- Convention used almost everywhere, and in this course: if a memory has *n* cells, the addresses of these cells will be from **0** to *n-1*.

Address Spaces For Instructions and Data

- Typically memory can be accessed using a single address space
  - For example, if we have 4 GB of memory, each byte has an address from 0 to 2^{32} - 1.
  - Each memory location may store instructions at some point and data at some other point
- An alternative is to have separate address spaces for instructions and data
  - In that case, a memory location is permanently dedicated to either storing instructions or to storing data
  - Instead of a single **load** instruction, we have **load_instructions** and **load_data**

Effects of Separate Address Spaces

- If A is a valid memory address, load_instructions A and load_data A access different memory locations.
  - load_instructions A accesses address A in the instructions space.
  - load_data A accesses address A in the data space.
- This makes it harder for malware to cause trouble. Why?

Effects of Separate Address Spaces

- If A is a valid memory address, load_instructions A and load_data A access different memory locations.
  - load_instructions A accesses address A in the instructions space.
  - load_data A accesses address A in the data space.
- This makes it harder for malware to cause trouble. Why?
- A common way for malware to attack is to:
  - Run as regular program.
  - Modify memory locations that store instructions, thus modifying other programs (such as the operating system).
- If instruction memory is accessed with different instructions, such behavior can easily be prevented.

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Stored Program Computers

Memory

- Accounting program (machine code)
- Editor program (machine code)
- C compiler (machine code)
- Payroll data
- Book text
- Source code in C for editor program

Processor
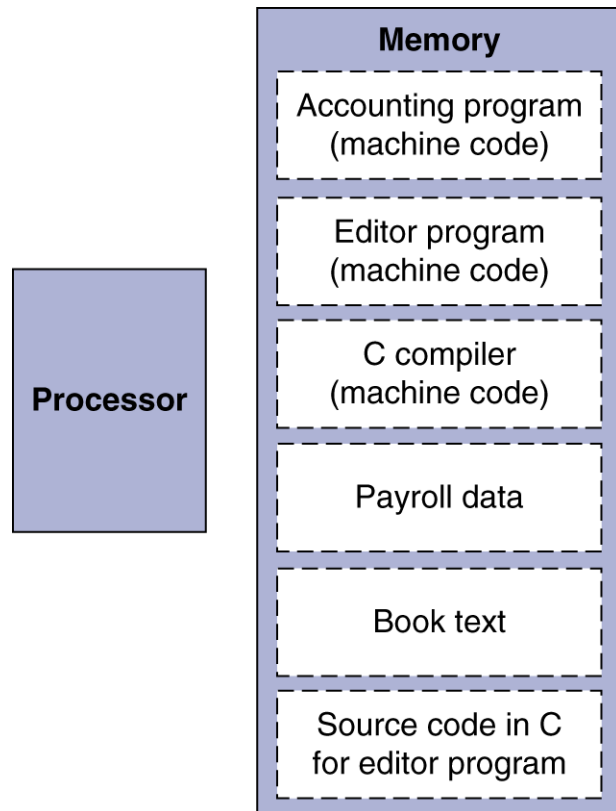
- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Questions?

?

Memory Operand Example 1

- C code:

$$g = h + A[8];$$

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:

  - Index 8 requires offset of 32

    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

| offset |

| base register |

Memory Operand Example 2

- C code:

A[12] = h + A[8];
  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)      # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)      # store word
```

Immediate Operands

- Constant data specified in an instruction

  `addi $s3, $s3, 4`

- No subtract immediate instruction
  - Just use a negative constant

    `addi $s2, $s1, -1`

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- ARM (and MIPS) instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Instruction fields**
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add $t0, $s1, $s2

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

ARMR-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

ARM Instructions in Machine Language

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bit	4 bits	1 bit	6 bits	6 bits	12 bits

- Opcode: Basic operation of the instruction
- Rd: The register destination operand. It gets the result of the operation
- Rn: The first register source operand
- Operand2:   The second source operand
- I: Immediate. If I is 0, the second source operand is a register. If I   is 1, the second source operand is a 12-bit immediate
- S: Set Condition Code. This field is related to conditional branch instructions
- Cond: Condition. Related to conditional branch instructions
- F: Instruction Format. This field allows ARM to different instruction formats when needed

ARM Instructions in Machine Language

Instruction	Format	Cond	F	I	op	S	Rn	Rd	Operand2
ADD	DP	14	0	0	4_{ten}	0	reg	reg	reg
SUB (subtract)	DP	14	0	0	$2s_{ten}$	0	reg	reg	reg
ADD (immediate)	DP	14	0	1	4_{ten}	0	reg	reg	constant
LDR (load word)	DT	14	1	n.a.	24_{ten}	n.a.	reg	reg	address
STR (store word)	DT	14	1	n.a.	25_{ten}	n.a.	reg	reg	address

- "**reg**" means a register number between 0 and 15 "**constant**" means a 12-bit constant

- "**address**" means a 12-bit address

- "**n.a.**" (not applicable) means this field does not appear in this format

- **Op** stands for opcode.

Logical Operations

Instructions for bitwise manipulation

Useful for extracting and inserting groups of bits in a word

Logical operations	C operators	Java operators	ARM instructions
Bit-by-bit AND	&	&	AND
Bit-by-bit OR	|	|	ORR
Bit-by-bit NOT	~	~	MVN
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR

C and Java logical operators and their corresponding ARM instructions.

ARM implements **NOT** using a **NOR** with one operand being zero.

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - sll by i bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by i bits divides by 2^i (unsigned only)

Questions?

?
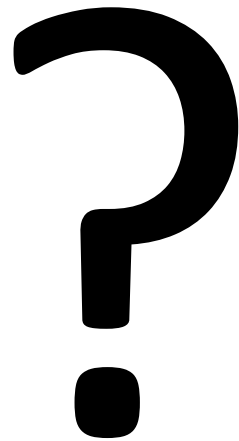
Shift Operations (for ARM)

Cond	F	I	opcode	S	Rn	Rd	Shift_imm		Shift		Rm
							Rs	0	Shift		Rm
14	0	0	4	0	2	5	2		0	0	5
14	0	0	13	0	0	6	4		1	0	5
14	0	0	13	0	0	6	3	0	1	1	5

ARM allows shifting by the value found in a register. The following instruction shifts register r5 right by the amount in register r3 and places the result in r6.

MOV   r6, r5,   LSR r3   ; r6 = r5 >> r3

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

$t2 | 0000 0000 0000 0000 0000 1101 1100 0000

$t1 | 0000 0000 0000 0000 0011 1100 0000 0000

$t0 | 0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0

- MIPS has NOR 3-operand instruction
  - a **NOR** b   ==   NOT (a **OR** b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

| $t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |

| $t0 | 1111 1111 1111 1111 1100 0011 1111 1111 |

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Conditional Operations

ARM-7:

This pair of instructions means go to the statement labeled L1 if the value in register1 equals the value in register2.

The mnemonic CMP stands for *compare* and BEQ stands for *branch if equal*.

```
CMP register1, register2
BEQ L1
```

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …

- Compiled MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses