CS2006: 計算機組織

Instruction Set Architecture

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

What Is Computer Architecture?

Computer Architecture =
Instruction Set Architecture
+ Machine Organization

- **"... the attributes of a [computing] system as seen by the [<u>assembly</u>] language] programmer, *i.e.* the conceptual structure and functional behavior ..."**

What are specified?

Recall in C Language

- **Operators: +, -, *, /, % (mod), ...**
  - **7/4==1, 7%4==3**
- **Operands:**
  - **Variables: `lower`, `upper`, `fahr`, `celsius`**
  - **Constants: `0`, `1000`, `-17`, `15.4`**
- **Assignment statement:**

  variable = expression

  - **Expressions consist of operators operating on operands, ex:**

```
celsius = 5*(fahr-32)/9;
a = b+c+d-e;
```

When Translating to Assembly ...

a = b + 5;

```
load    $r1, M[b]
load    $r2, 5
add     $r3, $r1, $r2
store   $r3, M[a]
```

Statement

Constant (immediate)

Memory

Register

Operands

Operator (op code)

Components of an ISA

- **Organization of programmable storage**
  - **registers**
  - **memory: flat, segmented**
  - **Modes of addressing and accessing data items and instructions**
- **Data types and data structures**
  - **encoding and representation (Chapter 3)**
- **Instruction formats**
- **Instruction set (or operation code)**
  - **ALU, control transfer, exceptional handling**

MIPS ISA as an Example

- **Instruction categories:**
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special

Registers

$r0 - $r31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

OP	$rs	$rt	$rd	sa	funct

OP	$rs	$rt	immediate

OP	jump target

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Operations of Hardware

- Syntax of basic MIPS arithmetic/logic instructions:

```
      1    2    3    4
   add $s0,$s1,$s2  # s0 = s1 + s2
```

  1) operation by name
  2) operand getting result ("destination")
  3) 1st operand for operation ("source1")
  4) 2nd operand for operation ("source2")
- Each instruction is 32 bits
- Syntax is rigid: 1 operator, 3 operands
  - Why? Keep hardware simple via regularity
- Design Principle 1: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Example

- How to do the following C statement?

```
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
```

Operands and Registers

- Unlike high-level language, assembly don't use variables
  → assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations are performed on these
- Benefits:
  - Registers in hardware → much faster than memory
  - Registers are easier for a compiler to use
    - Ex: as a place for temporary storage
  - Registers can hold variables to reduce memory traffic and improve code density (since register named with fewer bits than memory location) Ex: 5 bits vs. 32 bits

MIPS Registers

- 32 registers, each is 32 bits wide
  - Why 32? *Design Principle 2*: <u>Smaller is faster</u>
  - Groups of 32 bits called a ***word*** in MIPS
  - Registers are numbered from 0 to 31
  - Each can be referenced by number or name
  - Number references:

    `$0, $1, $2, … $30, $31`
  - By convention, each register also has a name to make it easier to code, ex:

    `$16 - $23` → `$s0 - $s7` (C variables)

    `$8 - $15` → `$t0 - $t7` (temporary)
- 32 x 32-bit FP registers (paired DP)
- Others: HI, LO, PC

Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

Fig. 2.18

MIPS R2000 Organization

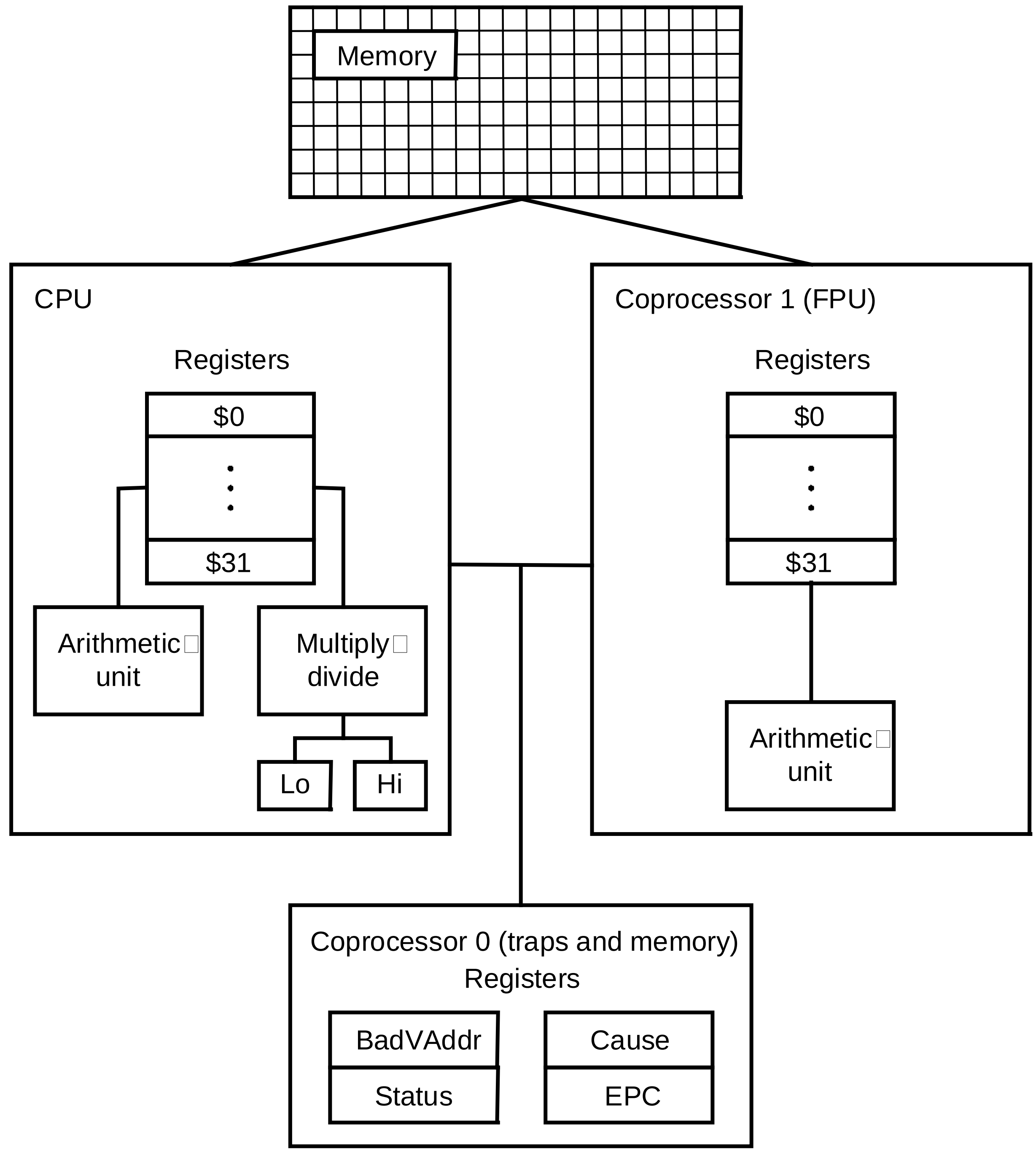


Fig. A.10.1

Example

- How to do the following C statement?

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

- `f`: `$s0`, `g`: `$s1`, `h`: `$s2`, `i`: `$s3`, `j`: `$s4`
- use intermediate temporary register `t0,t1`

```
add $t0,$s1,$s2   # t0 = g + h
add $t1,$s3,$s4   # t1 = i + j
sub $s0,$t0,$t1   # f=(g+h)-(i+j)
```

Register Architecture

- **Accumulator (1 register):**

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

  1+x address: addx A # acc $\leftarrow$ acc + mem[A+x]

- **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: (a special case of GPR)**

  3 address: add $ra,$rb,$rc # $ra $\leftarrow$ $rb + $rc

  load $ra,$rb # $ra $\leftarrow$ mem[$rb]

  store $ra,$rb # mem[$rb] $\leftarrow$ $ra

Register Organization Affects Programming

Code for C = A + B for four register organizations:

Stack	Accumulator	Register (reg-mem)	Register (load-store)
Push A	Load A	Load $r1,A	Load $r1,A
Push B	Add B	Add $r1,B	Load $r2,B
Add	Store C	Store C,$r1	Add $r3,$r1,$r2
Pop C			Store C,$r3

→ Register organization is an attribute of ISA!

Comparison: Byte per instruction? Number of instructions? Cycles per instruction?

Since 1975 all machines use GPRs

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Memory Operands

- C variables map onto registers; what about large data structures like arrays?
  - Memory contains such data structures
- However, MIPS arithmetic instructions operate on registers instead of directly on memory
  - Data transfer instructions (lw, sw, ...) to transfer between memory and register
  - A way to address memory operands

Data Transfer: Memory to Register (1/2)

- To transfer a word of data, need to specify two things:
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult
    - Think of memory as a 1D array
    - Address it by supplying a pointer to a memory address
    - Offset (in bytes) from this pointer
    - The desired memory address is the sum of these two values, ex: `8($t0)`
    - Specifies the memory address pointed to by the value in `$t0`, plus 8 bytes (why "bytes", not "words"?)
    - Each address is 32 bits

Data Transfer: Memory to Register (2/2)

- **Load Instruction Syntax:**

```
 1    2     3    4
lw $t0,12($s0)
```

  1) operation name
  2) register that will receive value
  3) numerical offset in bytes
  4) register containing pointer to memory
- **Example:** `lw $t0,12($s0)`
  - lw (Load Word, so a word (32 bits) is loaded at a time)
  - Take the pointer in `$s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`
- **Notes:**
  - `$s0` is called the ***base register***, 12 is called the ***offset***
  - Offset is generally used in accessing elements of array: base register points to the beginning of the array

Data Transfer: Register to Memory

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- Example: `sw $t0,12($s0)`
  - sw (Store Word, so a word (32 bits) is stored at a time)
  - This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into the memory address pointed to by the calculated sum

Compilation with Memory

- Compile by hand using registers:
  `$s1`:g, `$s2`:h, `$s3`:base address of `A`, A[ ] is integer

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

- What offset in `lw` to select an array element `A[8]` in a C program?
  - 4x8=32 bytes to select `A[8]`, sizeof(integer) = 4
  - 1st transfer from memory to register:

```
lw    $t0,32($s3)      # $t0 gets A[8]
```

  - Add 32 to `$s3` to select `A[8]`, put into `$t0`

- Next add it to h and place in g

```
add $s1,$s2,$t0      # $s1 = h+A[8]
```

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**
  - **Index 12 requires offset of 48**

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

Addressing: Byte versus Word

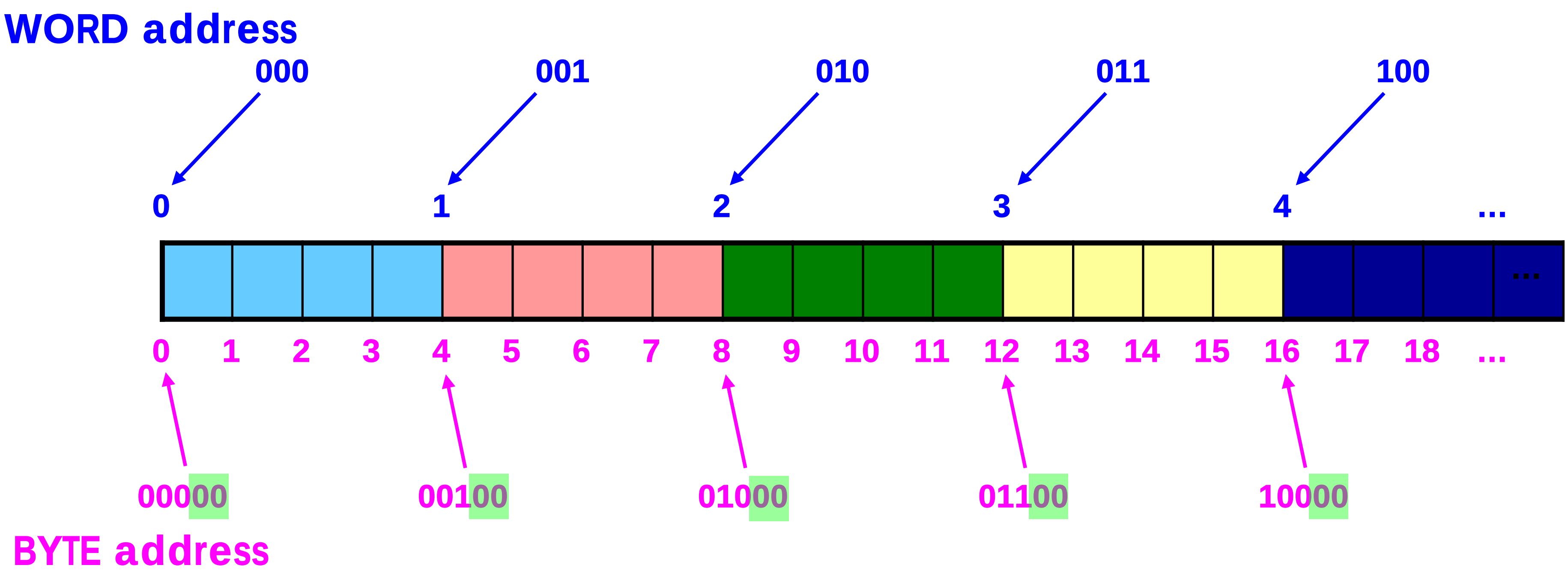
- **Every word in memory has an address, similar to an index in an array**
- **Early computers numbered words like C numbers elements of an array:**
  - **Memory[0], Memory[1], Memory[2], ...**

Called the "**address**" of a word

- **Computers need to access 8-bit bytes as well as words (4 bytes/word)**
- **Today, machines address memory as bytes, hence word addresses differ by 4**
  - **`Memory[0], Memory[4], Memory[8], ...`**
  - **This is also why lw and sw use bytes in offset**

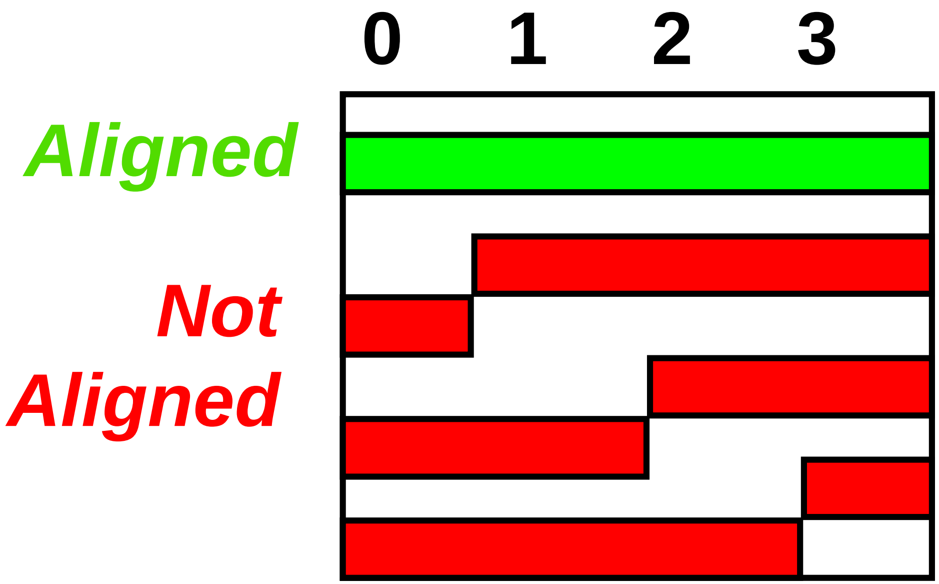
Addressing: Byte versus Word

- For each word, byte address is 4X of word address
  - The last 2 bits in byte address of a legal word are 0



A Note about Memory: Alignment

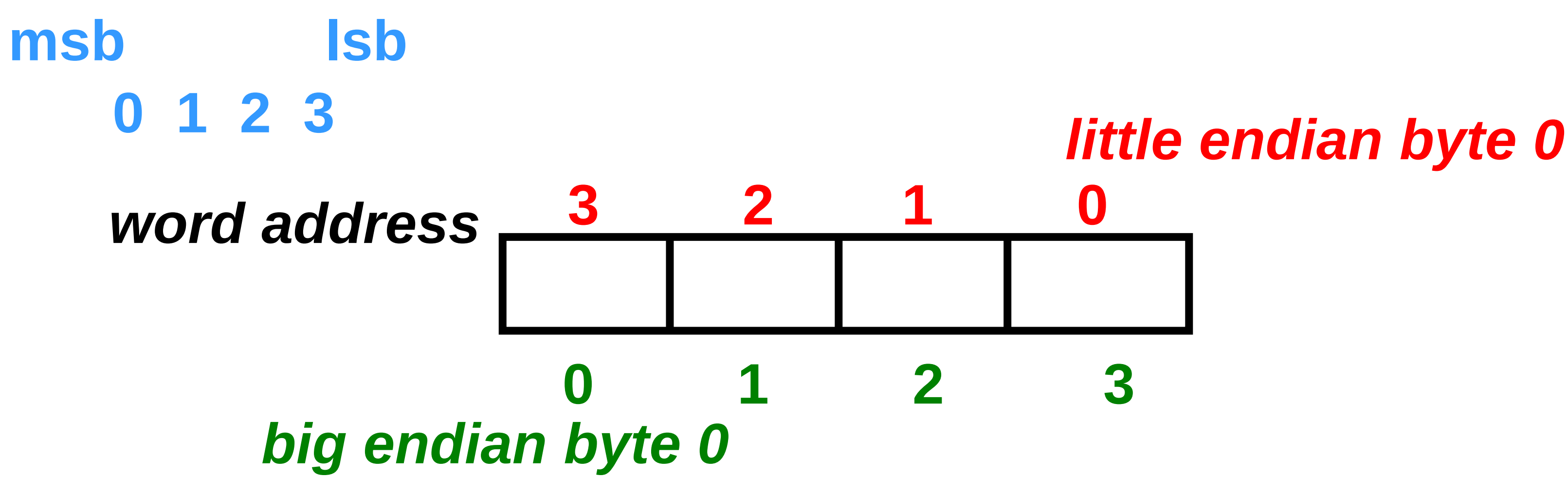
- MIPS requires that all words start at addresses that are multiples of 4 bytes



- Called Alignment: objects must fall on address that is multiple of their size

Another Note: Endianness

- **Byte order: numbering of bytes within a word**
- **Big Endian: address of most significant byte = word address**
  - **IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA**
- **Little Endian: address of least significant byte = word address**
  - **Intel 80x86, DEC Vax, DEC Alpha (Windows NT)**



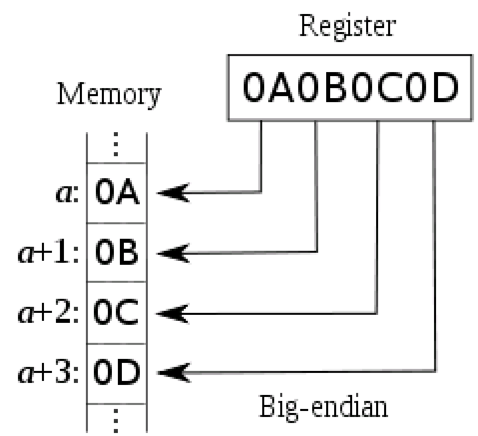
Endianness Example

- Assume the value to be stored in memory is "0x0A0B0C0D"

- The big endian:

increasing addresses →

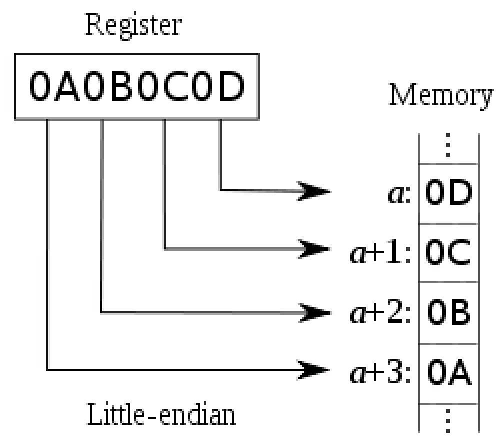
. . .	0x0A	0x0B	0x0C	0x0D	. . .



- The little endian:

increasing addresses →

. . .	0x0D	0x0C	0x0B	0x0A	. . .



Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variables in registers
  - Writes less commonly used variables to memory: <u>spilling</u>
- Why not keep all variables in memory?
  - <u>Smaller is faster</u>:
    registers are faster than memory
  - Registers are more versatile:
    - MIPS arithmetic instructions can read 2 registers, operate on them, and write 1 per instruction
    - MIPS data transfers only read or write 1 operand per instruction, and no operation

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Constants

- **Small constants used frequently (50% of operands)**
  ex: A = A + 5;
  B = B + 1;
  C = C - 18;
- **Put "typical constants" in memory and load them**
- **Constant data specified in an instruction:**
  addi \$29, \$29, 4
  slti \$8, \$18, 10
  andi \$29, \$29, 6
  ori \$29, \$29, 4
- ***Design Principle 3*: <u>Make the common case fast</u>**
  Which format?

Immediate Operands

- Immediate: numerical *constants*
  - Often appear in code, so there are special instructions for them
  - Add Immediate:

```
f = g + 10                 (in C)
addi $s0,$s1,10            (in MIPS)
```

    where `f`, `g` are in `$s0`, `$s1`
  - Syntax similar to add instruction, except that last argument is a number instead of a register
  - No subtract immediate instruction
    - Just use a negative constant

```
addi $s2, $s1, -1
```

The Constant Zero

- The number zero (0), appears very often in code; so we define register zero
- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
  - This is defined in hardware, so an instruction like `"addi $0,$0,5"` will not do anything
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero      # move $s1 to $t2
```

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Unsigned Number

- **In decimal system, we use 0-9 to represent numbers**

D =

d_3	d_2	d_1	d_0

$$D = d_3 \times 10^3 + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

$$0 \le D \le (9999)_{10}$$

- **In binary system, we use 0,1 to represent numbers**

B =

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$B = b_7 \times 2^7 + b_6 \times 2^6 + b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

$$0 \le B \le (11111111)_2 = 2^8 - 1$$

Signed Number

- To represent negative numbers, we use…
  - Negative sign '–'
  - Ex: -9_{10}, $-(1001)_2$,…
- The most critical question is…
  - How to store the negative sign in a computer?

Use one bit to store the negative sign
- Signed magnitude

Use logic complement for negative numbers
- 1's complement

Signed Magnitude

- Let the most signification bit as the sign bit

NO.	Binary	Unsigned	Signed Magnitude
7	111	+7	-3
6	110	+6	-2
5	101	+5	-1
4	100	+4	-0
3	011	+3	+3
2	010	+2	+2
1	001	+1	+1
0	000	+0	+0

1's Complement

- Bitwise inverse of the number as its negative number

NO.	Binary	Unsigned	Signed Magnitude	1's
7	111	+7	-3	-0
6	110	+6	-2	-1
5	101	+5	-1	-2
4	100	+4	-0	-3
3	011	+3	+3	+3
2	010	+2	+2	+2
1	001	+1	+1	+1
0	000	+0	+0	+0

Signed Number

- Both signed magnitude and 1's complement are not quite suitable for representing the numbers, since…
- 2 zeros (positive zero, negative zero)
- A special adder is required to perform addition
- Ex: 1 + (-1) = 0
  - Signed Magnitude: 001 + 101 = 110 (-2)
  - 1's Complement: 001 + 110 = 111 (-0)
- We need a GOOD representation for signed number

Odometer (1/2)

- Please design a 2-digit odometer (mileage meter)

00	01	02	03	...	96	97	98	99

$\longrightarrow$

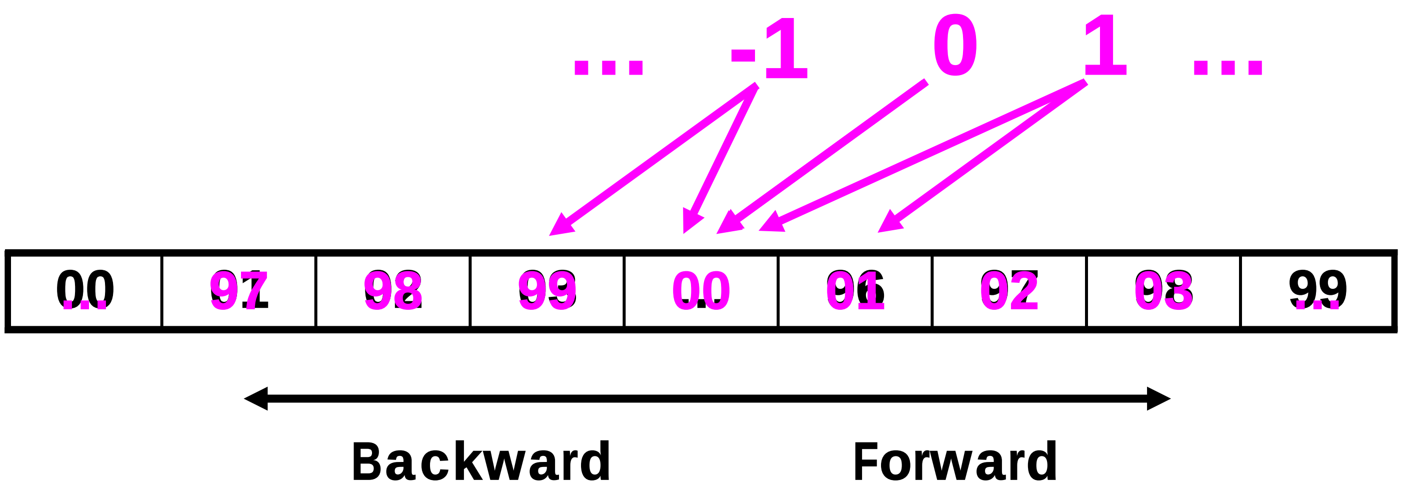
Forward

- Question1: 1 mile forward after 99?
  - 99+1 → 00, ignore the carry out

- Question2: 1 mile backward from 00?
  - 00 – 1 → 99, automatic borrow

Odometer (2/2)

- Use the odometer to represent negative mileage



- Where is the new zero?
  - 49? 50? 51? ← Definitely a bad Idea! We want to use 00
- Since 99+1 = 00, let's rotate the numbers!

Binary Odometer

- 2 possible representations...

All positive numbers begin in 0

All negative numbers begin in 1

011	+3
010	+2
001	+1
000	0
111	-1
110	-2
101	-3
100	-4

100	+4
011	+3
010	+2
001	+1
000	0
111	-1
110	-2
101	-3

- Which one is better?

Insight of 2's Complement

3	=	0	1	1		3	+	0	=	3
2		0	1	0		2		0		2
1		0	0	1		1		0		1
0		0	0	0		0		0		0
-1		1	1	1		3		-4		-1
-2		1	1	0		2		-4		-2
-3		1	0	1		1		-4		-3
-4		1	0	0		0		-4		-4

2's Complement

- MSB represents the negative number

B =

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$B = -b_7\times 2^7 + b_6\times 2^6 + b_5\times 2^5 + b_4\times 2^4 + b_3\times 2^3 + b_2\times 2^2 + b_1\times 2^1 + b_0\times 2^0$$

$$(10000000)_2 \le B \le (01111111)_2$$

$$-2^7 \le B \le (2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0) = 2^7-1$$

- For n-bit number
  - $-2^{n-1} \le B \le 2^{n-1}-1$

2's Complement

- 2's complement = 1's complement + 1

NO.	Binary	Unsigned	Signed Magnitude	1's	2's
7	111	+7	-3	-0	-1
6	110	+6	-2	-1	-2
5	101	+5	-1	-2	-3
4	100	+4	-0	-3	-4
3	011	+3	+3	+3	+3
2	010	+2	+2	+2	+2
1	001	+1	+1	+1	+1
0	000	+0	+0	+0	0

2's Complement Sign Extension

- **Assume there is a 3-bit integer**

$$A = \boxed{a_2 \mid a_1 \mid a_0}$$

$$A = -a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$$

- **How to store A in a 4-bit slot?**

$$A' = \boxed{a_3 \mid a_2 \mid a_1 \mid a_0}$$

$$A' = -a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$$

$$A' = A \rightarrow A' - A = 0$$

$$-a_3 \times 2^3 + a_2 \times 2^2 + a_2 \times 2^2 = -a_3 \times 2^3 + a_2 \times 2^3 = 0 \rightarrow a_3 = a_2$$

- **Sign Extension is done by duplicating the MSB**

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Instructions as Numbers

- **Currently we only work with words (32-bit blocks):**
  - **Each register is a word**
  - **`lw` and `sw` both access memory one word at a time**
- **So how do we represent instructions?**
  - **Remember: Computer only understands 1s and 0s, so "add `$t0,$0,$0`" is unknown to hardware**
  - **MIPS wants simplicity: since data is in words, make instructions be words…**

MIPS Instruction Format

- **One instruction is 32 bits**
  → divide instruction word into "fields"
  - **Each field tells computer something about instruction**
- **We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:**
  - ***R-format*: for register**
  - ***I-format*: for immediate, and `lw` and `sw` (since the offset counts as an immediate)**
  - ***J-format*: for jump**

R-Format Instructions (1/2)

- Define the following "fields":

6	5	5	5	5	6
`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`

  - opcode: partially specifies what instruction it is (Note: 0 for all R-Format instructions)
  - `funct`: combined with opcode to specify the instruction

    Question: Why not using a single field with 12 bits for opcode and `funct`?
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

R-Format Instructions (2/2)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31
  - Each of these fields specifies one of the 32 registers by number
- Shift amount:
  - `shamt`: contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
  - This field is set to 0 in all R-type instructions except the shift instructions

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

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

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

I-Format Instructions

- **Define the following "fields":**

6	5	5	16
`opcode`	`rs`	`rt`	`immediate`

  - **opcode: uniquely specifies an I-format instruction**
  - **`rs`: specifies the *only* register operand**
  - **`rt`: specifies register which will receive result of computation (*target register*)**
  - **`addi`, `slti`, immediate is sign-extended to 32 bits, and treated as a signed integer**
  - **16 bits → can be used to represent immediate up to 2^{16} different values**
- **Key concept: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location**

MIPS I-format Instructions

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

I-Format Example 1

- **MIPS Instruction:**

  ```
  addi   $21,$22,-50
  ```

  - **opcode = 8 (Figure 2.6)**
  - **rs = 22 (register containing operand)**
  - **rt = 21 (target register)**
  - **immediate = -50 (by default, this is decimal)**

decimal representation:

8	22	21	-50

binary representation:

001000	10110	10101	1111111111001110

I-Format Example 2

- **MIPS Instruction:**

  ```
  lw   $t0,1200($t1)
  ```

  - **opcode = 35 (Figure 2.6)**
  - **rs = 9 (base register)**
  - **rt = 8 (destination register)**
  - **immediate = 1200 (offset)**

decimal representation:

35	9	8	1200

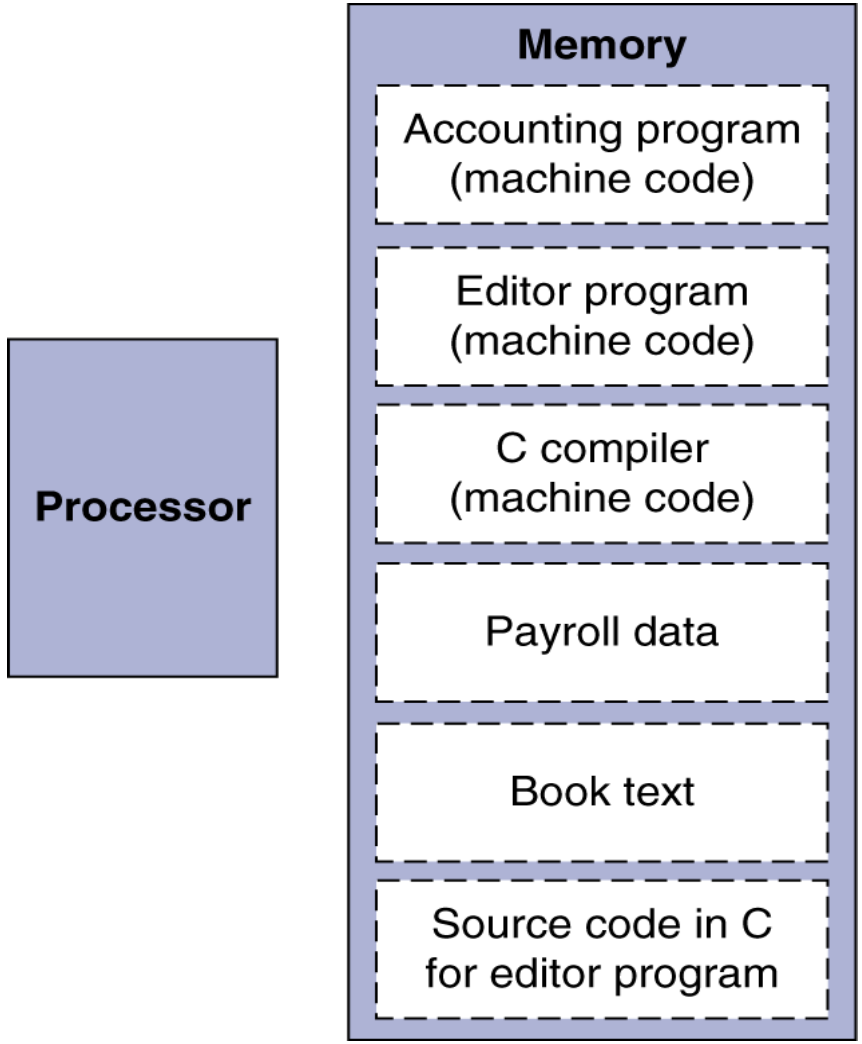
binary representation:

100011	01001	01000	0000010010110000

Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers
  2) Thus, entire programs can be stored in memory to be read or written just like numbers (data)
- One consequence: everything addressed
  - Everything has a memory address: instructions, data
    - both branches and jumps use addresses
  - One register keeps address of the instruction being executed: "Program Counter" (PC)
    - Basically a pointer to memory: Intel calls it Instruction Address Pointer, which is better
  - A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), etc.

Stored-Program Concept



Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Bitwise Operations

- Up to now, we've done arithmetic (add, sub, addi) and memory access (`lw` and `sw`)
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- New perspective: View contents of register as 32 bits rather than as a single 32-bit number
- Since registers are composed of 32 bits, we may want to access individual bits rather than the whole.
- Introduce two new classes of instructions:
  - Shift Instructions
  - Logical Operators

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

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)**

Shift Instructions (1/3)

- **Shift Instruction Syntax:**

```
    1      2    3   4
   sll   $t2,$s0,4
```

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) shift amount (constant)

- **MIPS has three shift instructions:**
  - `sll` (shift left logical): shifts left, fills empties with 0s
  - `srl` (shift right logical): shifts right, fills empties with 0s
  - `sra` (shift right arithmetic): shifts right, fills empties by "sign extension"

Shift Instructions (2/3)

- Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
- Example: "shift right logical" (srl) by 8 bits

$\boxed{\text{0001 0010 0011 0100 0101 0110}}$ 0111 1000

0000 0000 $\boxed{\text{0001 0010 0011 0100 0101 0110}}$

- Example: "shift left logical" (sll) by 8 bits

0001 0010 $\boxed{\text{0011 0100 0101 0110 0111 1000}}$

$\boxed{\text{0011 0100 0101 0110 0111 1000}}$ 0000 0000

Shift Instructions (3/3)

- **Example: "shift right arithmetic" (sra) by 8 bits**

  0001 0010 0011 0100 0101 0110 0111 1000

  0000 0000 0001 0010 0011 0100 0101 0110

- **Example: "shift right arithmetic" (sra) by 8 bits**

  1001 0010 0011 0100 0101 0110 0111 1000

  1111 1111 1001 0010 0011 0100 0101 0110

Uses for Shift Instructions (1/2)

- **Suppose we want to get byte 1 (bit 15 to bit 8) of a word in `$t0`. We can use:**

```
sll   $t0,$t0,16
srl   $t0,$t0,24
```

0001 0010 0011 0100 0101 0110 0111 1000

0101 0110 0111 1000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0101 0110

Uses for Shift Instructions (2/2)

- Shift for multiplication: in binary
  - Multiplying by 4 is same as shifting left by 2:
    - 11_2 x 100_2 = 1100_2
    - 1010_2 x 100_2 = 101000_2
  - Multiplying by 2^n is same as shifting left by n
- Since shifting is much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction (i.e. strength reduction):

```
a *= 8;                    (in C)
```

would be compiled to:

```
sll   $s0,$s0,3     (in MIPS)
```

Logical Operators

- Logical instruction syntax:

```
1      2    3    4
or     $t0, $t1, $t2
```

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) second operand (register) or immediate (numerical constant)
- Instruction names:
  - and, `or`: the third argument is a register
  - `andi`, `ori`: the third argument is an immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.

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

MIPS Logical Instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, zero exten.
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, zero exten.
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)

So Far...

- All instructions have allowed us to manipulate data.
- So we've built a calculator.
- In order to build a computer, we need the ability to make decisions...

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

MIPS Decision Instructions

```
beq   register1, register2, L1
```

- **Decision instruction in MIPS:**

  ```
  beq   register1, register2, L1
  ```
  "Branch if (registers are) equal"
  meaning :
  ```
   if  (register1==register2) goto L1
  ```
- **Complementary MIPS decision instruction**

  ```
  bne   register1, register2, L1
  ```
  "Branch if (registers are) not equal"
  meaning :
  ```
   if  (register1!=register2) goto L1
  ```
- **These are called conditional branches**

MIPS Goto Instruction

```
j     label
```

- MIPS has an unconditional branch:
  ```
  j     label
  ```
  - Called a Jump Instruction: jump directly to the given label without testing any condition
  - meaning :
    ```
    goto   label
    ```
- Technically, it's the same as:
  ```
  beq      $0, $0, label
  ```
  since it always satisfies the condition
- It uses the J-type instruction format

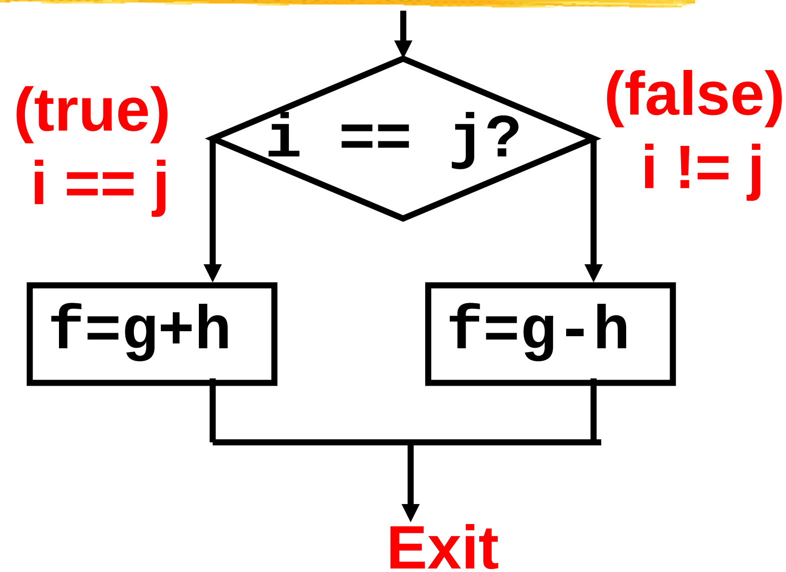
Compiling C if into MIPS

- **Compile by hand**

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

- **Use this mapping:**

  f: $s0, g: $s1, h: $s2,
  i: $s3, j: $s4



- **Final compiled MIPS code:**

```
        bne   $s3,$s4,Else    # branch i!=j
        add   $s0,$s1,$s2     # f=g+h(true)
        j     Exit            # go to Exit
Else:   sub   $s0,$s1,$s2     # f=g-h (false)
Exit:
```

Note: Compiler automatically creates labels to handle decisions (branches) appropriately

Compiling C Loop Statements

- **C code:**

  `while (save[i] == k) i += 1;`

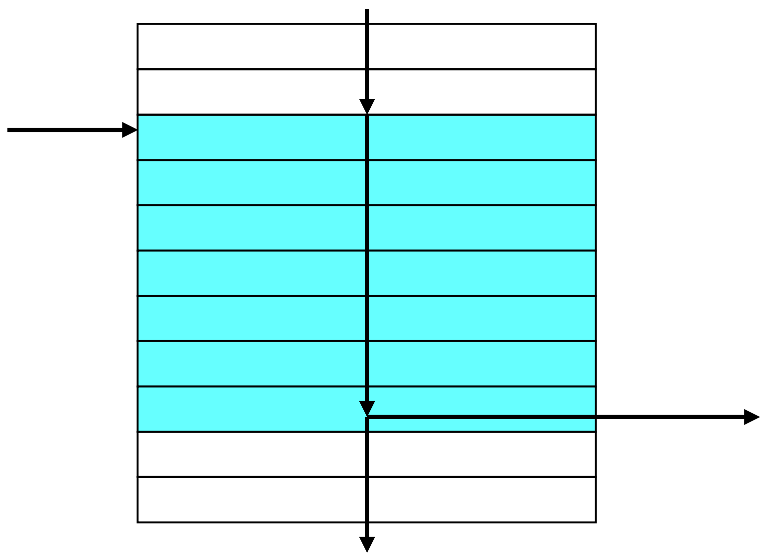
  - i: $s3, k: $s5, base address of save[]: $s6
  - save[] is integer, sizeof(integer) = 4

- **Compiled MIPS code:**

```
Loop:    sll  $t1, $s3, 2      #$t1=i x 4
         add  $t1, $t1, $s6    #$t1=addr of save[i]
         lw   $t0, 0($t1)      #$t0=save[i]
         bne  $t0, $s5, Exit   #if save[i]!=k goto Exit
         addi $s3, $s3, 1      #i=i+1
         j    Loop             #goto Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A **compiler** identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Inequalities in MIPS

- Until now, we've only tested equalities (== and != in C), but general programs need to test < and >
- Set on Less Than:
  - `slt $rd, $rs, $rt`
    - if ($rs < $rt) $rd = 1; else $rd = 0;
  - `slti $rt, $rs, constant`
    - if ($rs < constant) $rt = 1; else $rt = 0;

Compile by hand: `if (g < h) goto Less;`
Let g: `$s0`, h: `$s1`

```
slt $t0,$s0,$s1    # $t0 = 1 if g<h
bne $t0,$0,Less    # goto Less if $t0!=0
```

MIPS has no "branch on less than" → too complex

Inequality Example

C

```
    if (g >= 1) goto Loop
Loop:     . . .
```

MIPS

```
slti $t0,$s0,1     # $t0 = 1 if $s0<1 (g<1)
beq  $t0,$0,Loop   # goto Loop if $t0==0
```

Signed vs. Unsigned Comparisons

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`

```
slt  $t0, $s0, $s1  # signed
```

–1 < +1 ⇒ `$t0 = 1`

```
sltu $t0, $s0, $s1  # unsigned
```

+4,294,967,295 > +1 ⇒ `$t0 = 0`

Branch Instruction Design

- Why not `blt`, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction → a slower clock
  - All instructions penalized!
- beq and bne are the common cases
- This is a good design compromise

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

C Function Call Bookkeeping

```
sum = leaf_example(a,b,c,d) . . .

int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- **Return address** `$ra`
- **Procedure address** `Labels`
- **Arguments** **`$a0`, `$a1`, `$a2`, `$a3`**
- **Return value** **`$v0`, `$v1`**
- **Local variables** **`$s0`, `$s1`, …, `$s7`**

Note the use of register conventions

Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	

16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

Fig. 2.18

Procedure Call Instructions

- **Procedure call: jump and link**
  `jal ProcedureLabel`
  - **Address of following instruction put in `$ra`**
  - **Jumps to target address (i.e., `ProcedureLabel`)**
- **Procedure return: jump register**
  `jr $ra`
  - **Copies `$ra` to program counter**
  - **Can also be used for computed jumps**
    - **Ex: for case/switch statements**
      - **Jump table is an array of addresses corresponding to labels in codes**
      - **Load appropriate entry to register**
      - **Jump register**

Leaf Procedure Example

- **C code:**

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - **Arguments `g:$a0, h:$a1, i:$a2, j:$a3`**
  - **`f` in `$s0` (hence, we need to save `$s0` on stack)**
  - **`$t0` and `$t1` are not saved on stack**
  - **Result in `$v0`**

Leaf Procedure Example

- **MIPS code:**

```
leaf_example:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

Save $s0 on stack

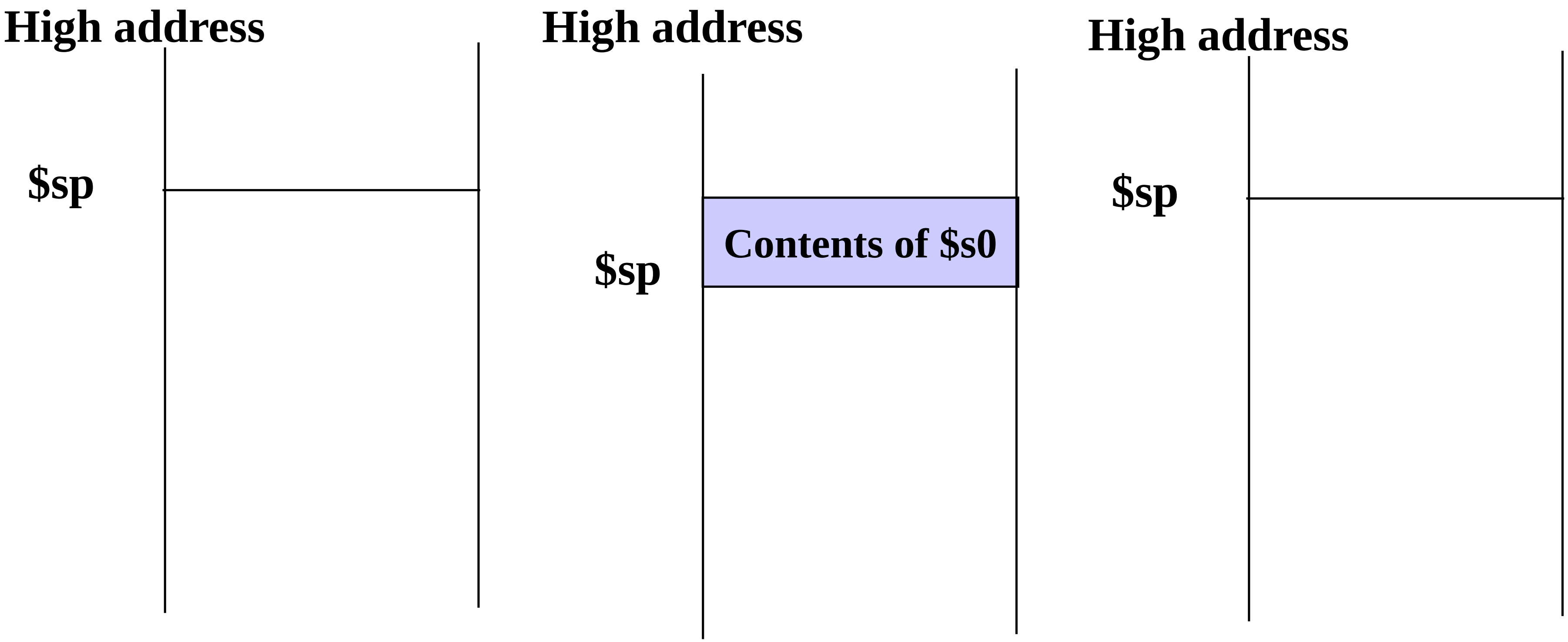
Procedure body

Result

Restore $s0

Return

Local Data on the Stack



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call (because callee will not save them)
- Restore from the stack after the call

Non-Leaf Procedure Example

- **C code:**

```c
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

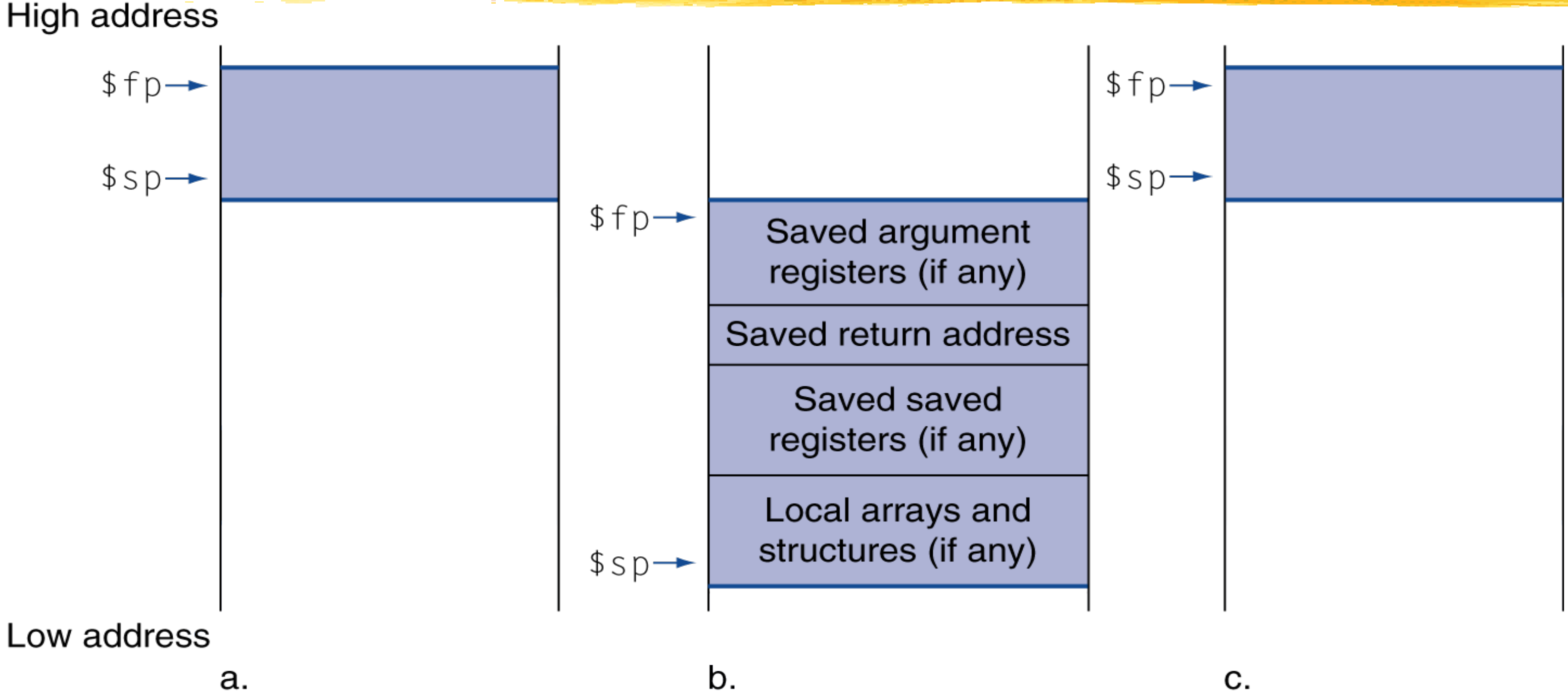
  - **Argument n in `$a0`**
  - **Result in `$v0`**

Non-Leaf Procedure Example

- **MIPS code:**

```
fact:
     addi $sp, $sp, -8      # adjust stack for 2 items
     sw   $ra, 4($sp)       # save return address
     sw   $a0, 0($sp)       # save argument
     slti $t0, $a0, 1       # test for n < 1
     beq  $t0, $zero, L1
     addi $v0, $zero, 1     # if so, result is 1
     addi $sp, $sp, 8       #   pop 2 items from stack
     jr   $ra               #   and return
L1:  addi $a0, $a0, -1      # else decrement n
     jal  fact              # recursive call
     lw   $a0, 0($sp)       # restore original n
     lw   $ra, 4($sp)       #   and return address
     addi $sp, $sp, 8       # pop 2 items from stack
     mul  $v0, $a0, $v0     # multiply to get result
     jr   $ra               # and return
```

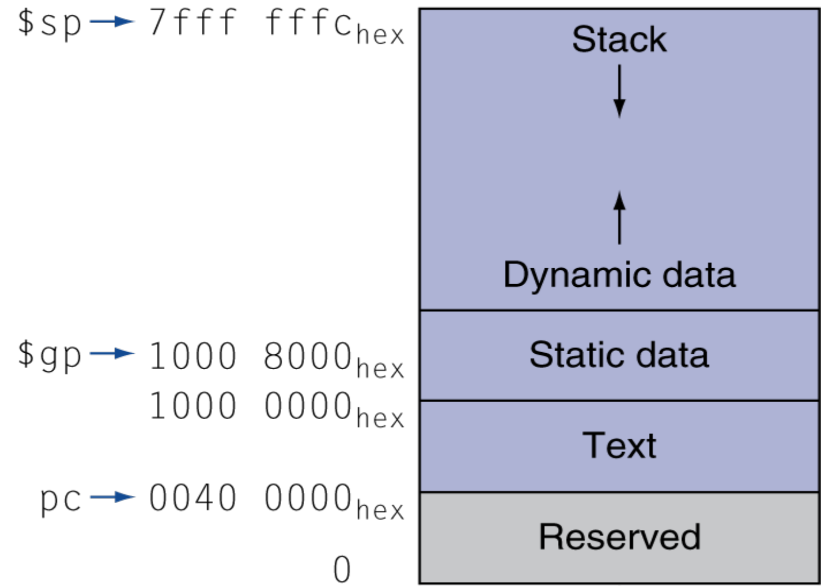
Local Data on the Stack



- Local data allocated by callee
  - Ex: C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - Ex: static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - Ex: `malloc()` in C, `new()` in C++/Java
- Stack: automatic storage



$GP, $SP, $FP

- $GP (Global Pointer)
  - The register that is reserved to point to static data
- $SP (Stack Pointer)
  - A value denoting the most recently allocated address in a stack
- $FP (Frame Pointer)
  - A value denoting the location of the saved registers and local variables for a given procedure

Why Procedure Conventions?

- Definitions
  - Caller: function making the call, using `jal`
  - Callee: function being called
- Procedure conventions as a contract between the Caller and the Callee
- If both the Caller and Callee obey the procedure conventions, there are significant benefits
  - People who have never seen or even communicated with each other can write functions that work together
  - Recursion functions work correctly

Caller's Rights, Callee's Rights

- **Callees' rights:**
  - Right to use VAT registers freely
  - Right to assume arguments are passed correctly
- **To ensure callees' right, caller saves registers:**
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - `$t` Registers `$t0 - $t9`
- **Callers' rights:**
  - Right to use `$s` registers without fear of being overwritten by callee
  - Right to assume return value will be returned correctly
- **To ensure caller's right, callee saves registers:**
  - `$s` Registers `$s0 - $s7`

Contract in Function Calls (1/2)

- **Caller's responsibilities (how to call a function)**
  - **Slide `$sp` down to reserve memory:**
    ex: `addi $sp, $sp, -28`
  - **Save `$ra` on stack because `jal` clobbers it:**
    ex: `sw $t0, 20 ($sp)`
    `sw $ra, 24 ($sp)`
  - **If still need their values after function call, save `$v`, `$a`, `$t` on stack or copy to `$s` registers**
  - **Put first 4 words of arguments in `$a0-3`, additional arguments go on stack: "a4" is `16($sp)`**
  - **`jal` to the desired function**
  - **Receive return values in `$v0, $v1`**
  - **Undo first steps: ex:**
    ```
    lw $t0, 20($sp)
    lw $ra, 24($sp)
    addi $sp, $sp, 28
    ```

Contract in Function Calls (2/2)

- **Callee's responsibilities (i.e. how to write a function)**
  - **If using `$s` or big local structures, slide `$sp` down to reserve memory, ex:**
    ```
    addi $sp, $sp, -48
    ```
  - **If using `$s`, save before using, ex:**
    ```
    sw $s0, 44($sp)
    ```
  - **Receive arguments in `$a0-3`, additional arguments on stack**
  - **Run the procedure body**
  - **If not void, put return values in `$v0,$v1`**
  - **If applicable, undo first two steps: ex:**
    ```
    lw $s0, 44($sp)
    addi $sp, $sp, 48
    ```
  - **`jr $ra`**

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
- String processing is a common case

- Sign extend to 32 bits in `$rt`

```
lb $rt, offset($rs)      lh $rt, offset($rs)
```

- Zero extend to 32 bits in `$rt`

```
lbu $rt, offset($rs)     lhu $rt, offset($rs)
```

- Store just rightmost byte/halfword

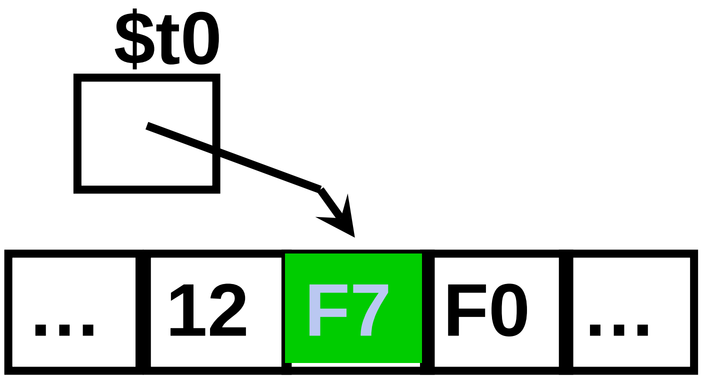
```
sb $rt, offset($rs)      sh $rt, offset($rs)
```

MIPS Data Transfer Instructions

Instruction		Comment
sw	\$t3,500(\$t4)	Store word
sh	\$t3,502(\$t2)	Store half
sb	\$t2,41(\$t3)	Store byte
lw	\$t1, 30(\$t2)	Load word
lh	\$t1, 40(\$t3)	Load halfword
lhu	\$t1, 40(\$t3)	Load halfword unsigned
lb	\$t1, 40(\$t3)	Load byte
lbu	\$t1, 40(\$t3)	Load byte unsigned
lui	\$t1, 40	Load Upper Immediate (16 bits shifted left by 16)

What does it mean?

Load Byte Signed/Unsigned



lb $t1, 0($t0)

$t1: FFFFFF F7 — Sign-extension

lbu $t2, 0($t0)

$t2: 000000 F7 — Zero-extension

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in `$a0`, `$a1`
  - `i` in `$s0`

String Copy Example

- **MIPS code:**

```
strcpy:
      addi $sp, $sp, -4         # adjust stack for 1 item
      sw   $s0, 0($sp)          # save $s0
      add  $s0, $zero, $zero    # i = 0
L1:   add  $t1, $s0, $a1        # addr of y[i] in $t1
      lbu  $t2, 0($t1)          # $t2 = y[i]
      add  $t3, $s0, $a0        # addr of x[i] in $t3
      sb   $t2, 0($t3)          # x[i] = y[i]
      beq  $t2, $zero, L2       # exit loop if y[i] == 0
      addi $s0, $s0, 1          # i = i + 1
      j    L1                   # next iteration of loop
L2:   lw   $s0, 0($sp)          # restore saved $s0
      addi $sp, $sp, 4          # pop 1 item from stack
      jr   $ra                  # and return
```

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

32-bit Constants

- **Most constants are small**
  - **16-bit immediate is sufficient**
- **For the occasional 32-bit constant**
- **Load Upper Immediate:**

  `lui $rt, constant`

  - **Copies 16-bit constant to left 16 bits of `$rt`**
  - **Clears right 16 bits of `$rt` to 0**

32-bit	constant

machine version of `lui`

001111 00000 10000	0000 0000 0011 1101

`$s0`

`lui $s0, 61`

0000 0000 0011 1101	0000 0000 0000 0000

`$s0`

`ori $s0, $s0, 2304`

0000 0000 0011 1101	0000 1001 0000 0000

Load Upper Immediate Example

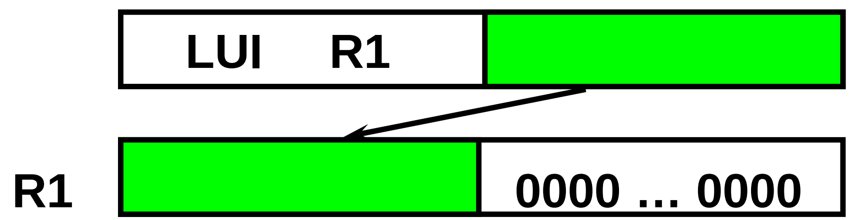
What if your constant is too big in your assembly code?

- Assembler automatically separate your immediate into two 16-bit halfwords
- Ex:

```
addi   $t0,$t0, 0xABABCDCD
```

becomes:

```
lui    $at, 0xABAB #$at:reserved for assembler
ori    $at, $at, 0xCDCD
add    $t0,$t0,$at
```



Branch Addressing (1/2)

- Use I-format:

opcode	rs	rt	immediate

  - opcode specifies beq or bne
  - Rs and Rt specify registers to compare
- What can *immediate* specify? PC-relative addressing
  - *Immediate* is only 16 bits, but PC is 32-bit
    → *immediate* cannot specify entire address
  - Loops are generally small: < 50 instructions
    - Though we want to branch to anywhere in memory, a single branch only need to change PC by a small amount
  - How to use PC-relative addressing
    - 16-bit *immediate* as a signed two's complement integer to be *added* to the PC if branch taken
    - Now we can branch +/- 2^{15} bytes from the PC ?

NOT GOOD ENOUGH !!!

Branch Addressing (2/2)

- *`Branch Immediate`* specifies **word address**
  - Instructions are word aligned (byte address is always a multiple of 4, i.e., it ends with 00 in binary)
    - The number of bytes to add to the PC will always be a multiple of 4
  - Specify the *`immediate`* in words (confusing?)
  - Now, we can branch +/- 2^{15} **words** from the PC (or +/- 2^{17} bytes), handle loops 4 times as large

- *`Immediate`* specifies **PC + 4**
  - Due to hardware, add `immediate` to (PC+4), not to PC
  - If branch not taken: PC = PC + 4
  - If branch taken: PC = (PC+4) + (`immediate`*4)

Branch Example

- **MIPS Code:**

```
Loop:  beq    $9,$0,End
       add    $8,$8,$10
       addi   $9,$9,-1
       j      Loop
End:   ...
```

- **Branch is I-Format:**

opcode	rs	rt	immediate

opcode = 4 (Figure 2.20)
`rs` = 9 (first operand)
`rt` = 0 (second operand)
`immediate` = ???

- **Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch**
  → `immediate` = 3

Branch Example

- **MIPS Code:**

```
   Loop:  beq    $9,$0,End
          add    $8,$8,$10
          addi   $9,$9,-1
          j      Loop
    End:
```

decimal representation:

4	9	0	3

binary representation:

000100	01001	00000	0000000000000011

Jump Addressing (1/3)

- For branches, we assumed that we don't want to branch too far, so we can specify change in PC.
- For general jumps (`j` and `jal`), we may jump to anywhere in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit instruction word, so we <u>compromise</u>.

Jump Addressing (2/3)

- Define "fields" of the following number of bits each:

6 bits	26 bits

- As usual, each field has a name:

opcode	target address

- **Key concepts:**
  - Keep opcode field identical to R-format and I-format for consistency
  - Combine other fields to make room for target address
- Optimization:
  - Jumps only jump to word aligned addresses
    - last two bits are always 00 (in binary)
    - specify 28 bits of the 32-bit bit address

Jump Addressing (3/3)

- **Where do we get the other 4 bits?**
  - **Take the 4 highest order bits from the PC**
  - **Technically, this means that we cannot jump to anywhere in memory, but it's adequate 99.9999...% of the time, since programs are not that long**
  - **Linker and loader avoid placing a program across an address boundary of 256 MB**
- **Summary:**
  - **New PC = PC[31..28] : target address (26 bits) : 00**
  - **Note: ':' means concatenation**
    4 bits : 26 bits : 2 bits = 32-bit address
- **If we absolutely need to specify a 32-bit address:**
  - **Use *jr $ra* # jump to the address specified by $ra**

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

`Loop: sll  $t1, $s3, 2`	80000	0	0	19	9	2	0
`      add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
`      lw   $t0, 0($t1)`	80008	35	9	8	0		
`      bne  $t0, $s5, Exit`	80012	5	8	21	2		
`      addi $s3, $s3, 1`	80016	8	19	19	1		
`      j    Loop`	80020	2	20000				
`Exit: …`	80024						

- $80016 + \boxed{2} \times 4 = 80024$
- $\boxed{20000} \times 4 = 80000$

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

L1: Short branching distance

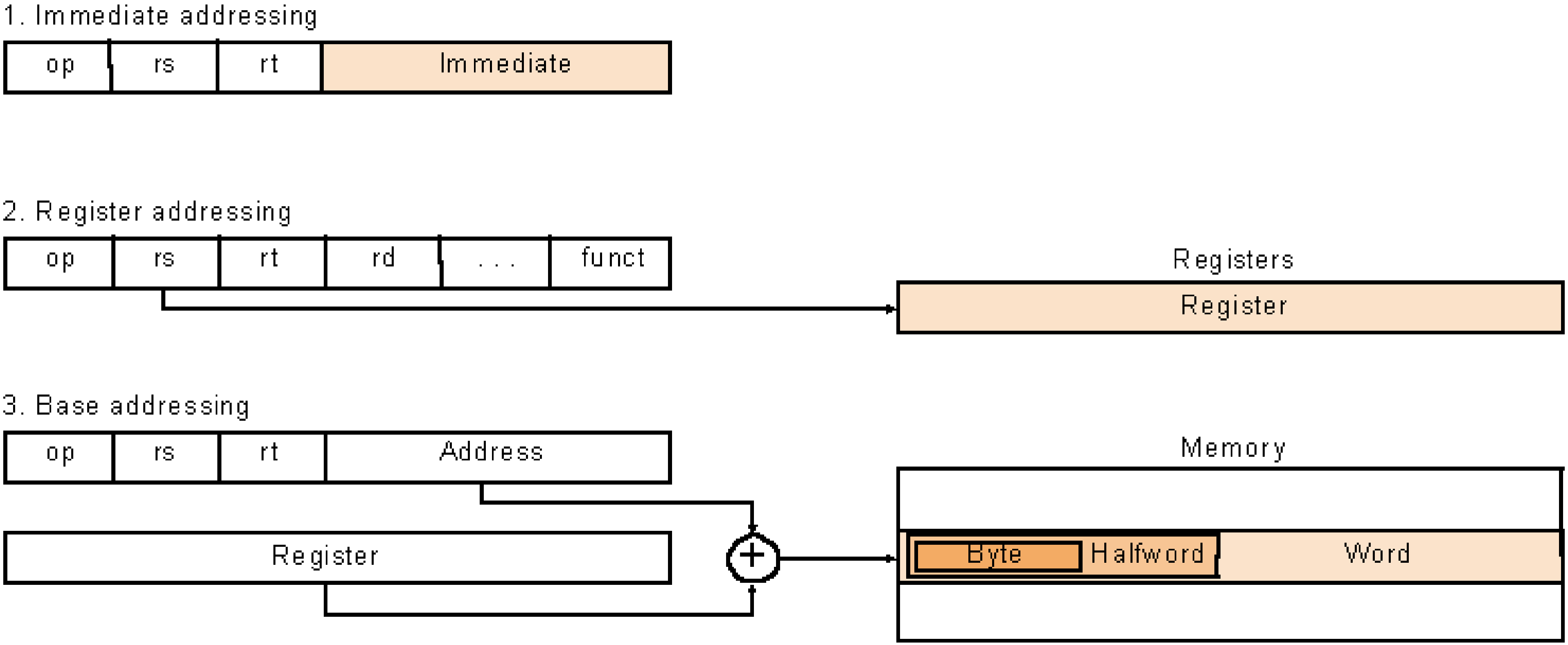
```
        beq $s0,$s1, L1
```

↓

L1: Long branching distance

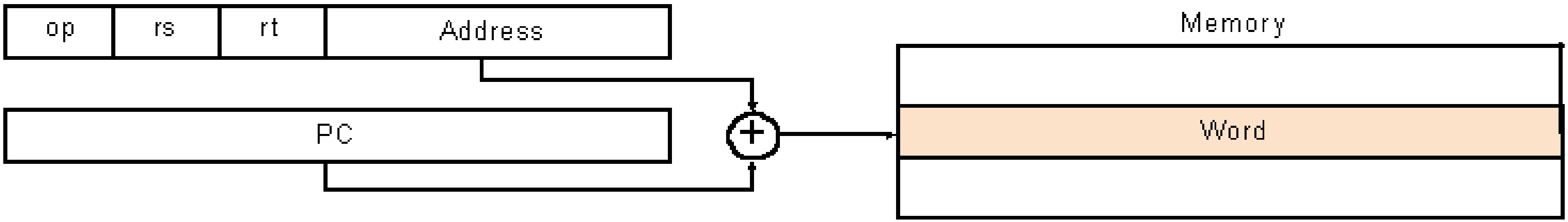
```
        bne $s0,$s1, L2
        j L1
L2:     …
```

MIPS Addressing Mode

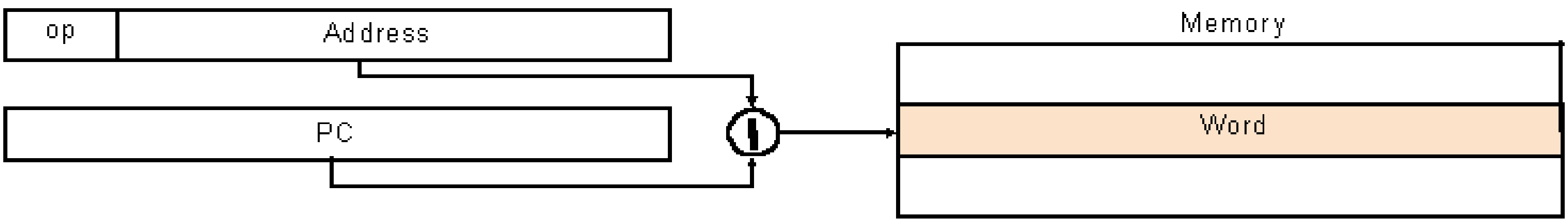


MIPS Addressing Modes

4. PC-relative addressing



5. Pseudodirect addressing



Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - Ex: atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

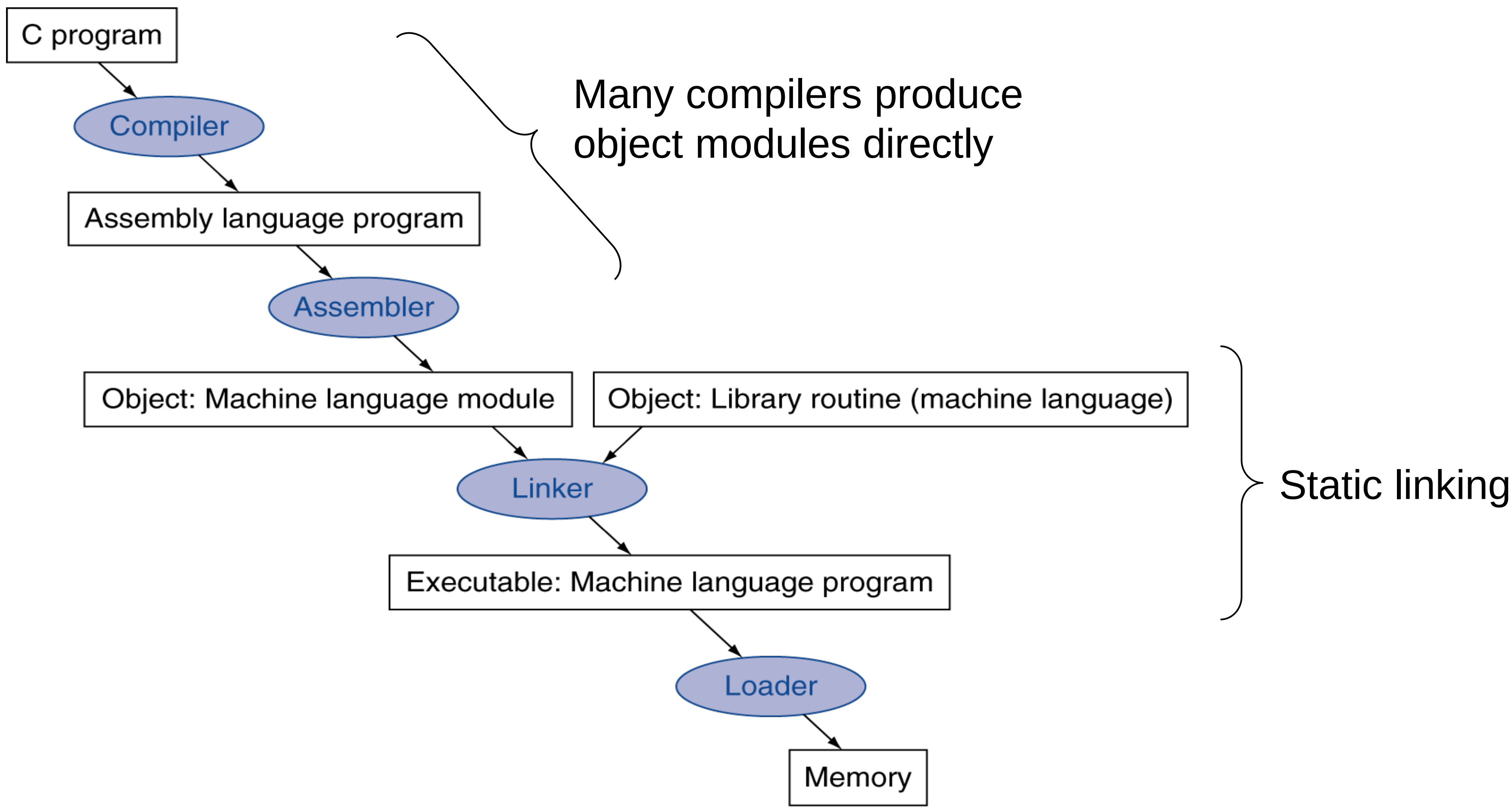
- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `$rt`
  - Fails if location is changed
    - Returns 0 in `$rt`
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4   ;copy exchange value
     ll  $t1,0($s1)      ;load linked
     sc  $t0,0($s1)      ;store conditional
     beq $t0,$zero,try   ;branch store fails
     add $s4,$zero,$t1   ;put load value in $s4
```

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Translation and Startup



Assembler Pseudoinstructions

- **Most assembler instructions represent machine instructions one-to-one**
- **Pseudo instructions: figments of the assembler's imagination**

```
move $t0, $t1      →   add $t0, $zero, $t1
blt $t0, $t1, L    →   slt $at, $t0, $t1
                       bne $at, $zero, L
```

  - **`$at` (register 1): assembler temporary**

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- **Produces an executable image**
  1. **Merges segments**
  2. **Resolve labels (determine their addresses)**
  3. **Patch location-dependent and external refs**
- **Could leave location dependencies for fixing by a relocating loader**
  - **But with virtual memory, no need to do this**
  - **Program can be loaded into absolute location in virtual memory space**

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including `$sp`, `$fp`, `$gp`)
  6. Jump to startup routine
     - Copies arguments to `$a0`, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

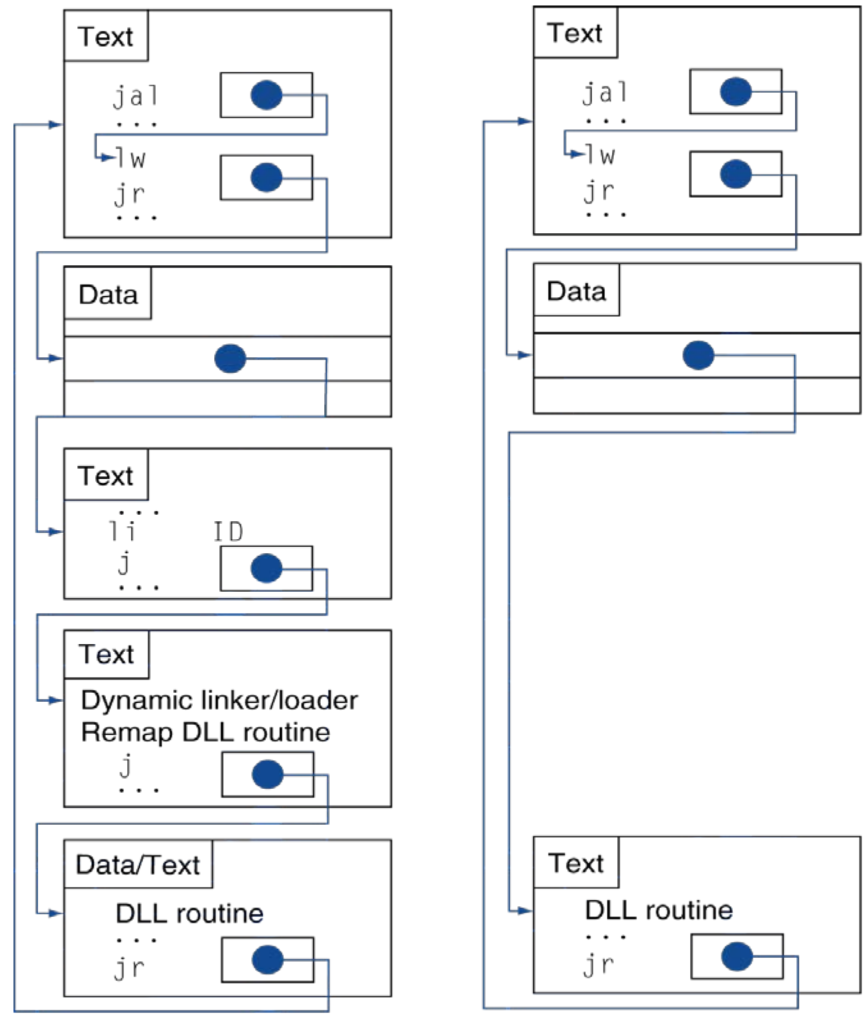
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

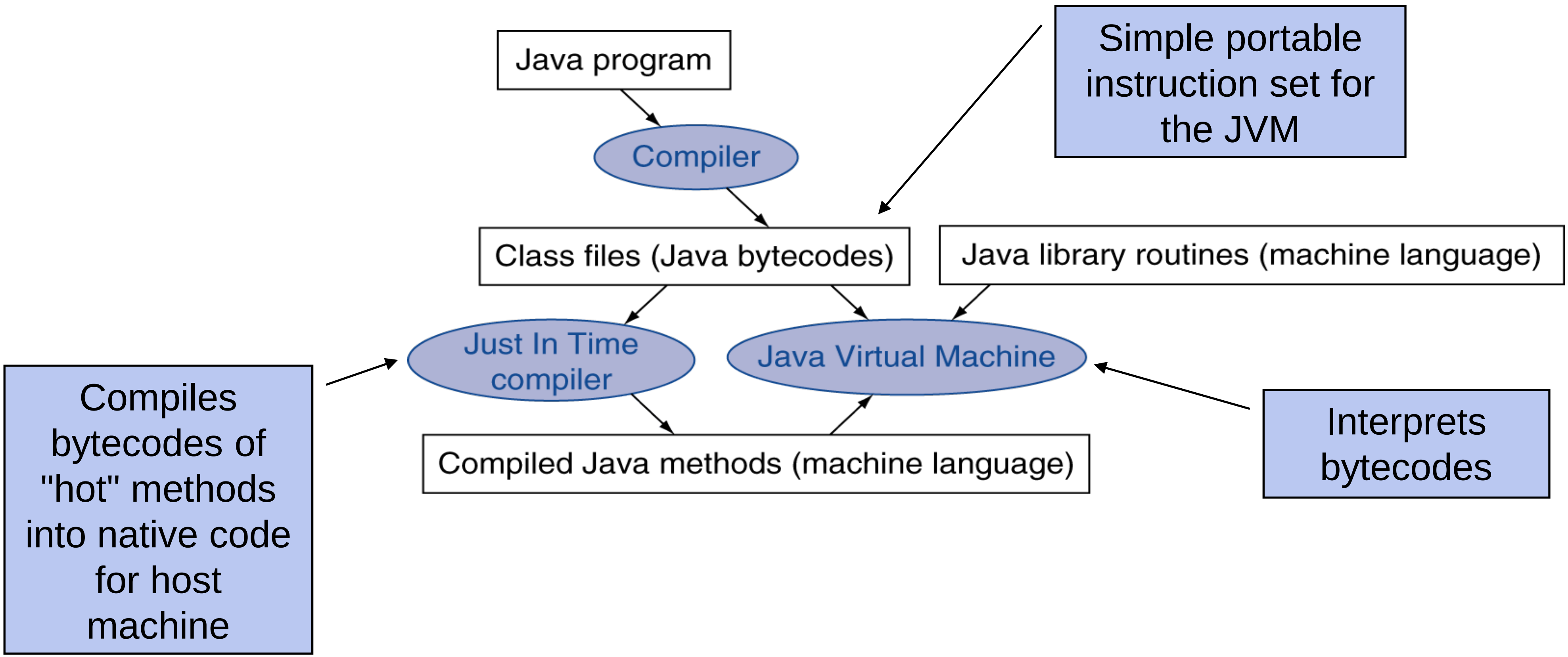
Dynamically mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting Java Applications



Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - Base of `v[]`:`$a0`, k:`$a1`, `temp`:`$t0`

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- **Non-leaf (calls swap)**

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
          j >= 0 && v[j] > v[j + 1];
          j -= 1) {
      swap(v,j);
    }
  }
}
```

  - **Base of v[]:$a0, n:$a1, i:$s0, j:$s1**

The Procedure Body

```
         move $s2, $a0           # save $a0 into $s2
         move $s3, $a1           # save $a1 into $s3
         move $s0, $zero         # i = 0
for1tst: slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, -1       # j = i - 1
for2tst: slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2        # $t1 = j * 4
         add  $t2, $s2, $t1      # $t2 = v + (j * 4)
         lw   $t3, 0($t2)        # $t3 = v[j]
         lw   $t4, 4($t2)        # $t4 = v[j + 1]
         slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
         move $a0, $s2           # 1st param of swap is v (old $a0)
         move $a1, $s1           # 2nd param of swap is j
         jal  swap               # call swap procedure
         addi $s1, $s1, -1       # j -= 1
         j    for2tst            # jump to test of inner loop
exit2:   addi $s0, $s0, 1        # i += 1
         j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

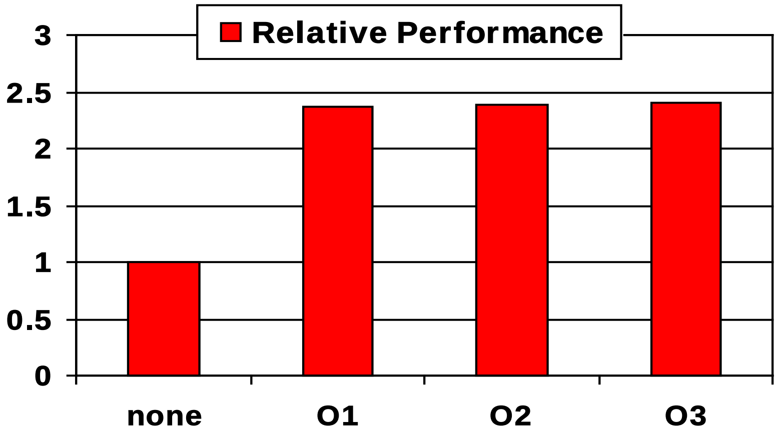
Outer loop

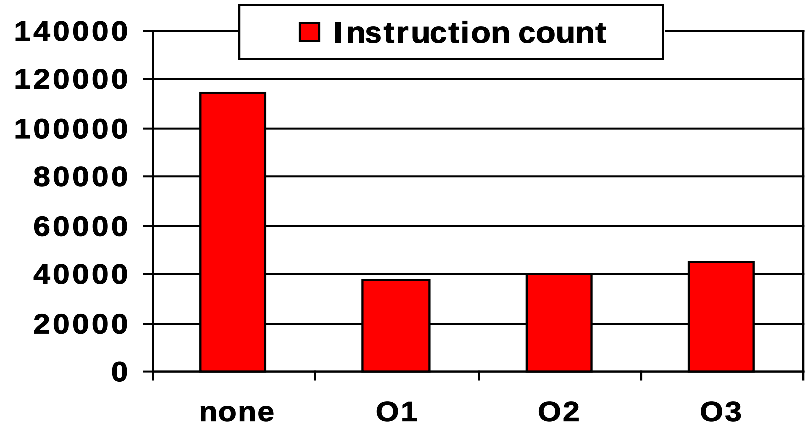
The Full Procedure

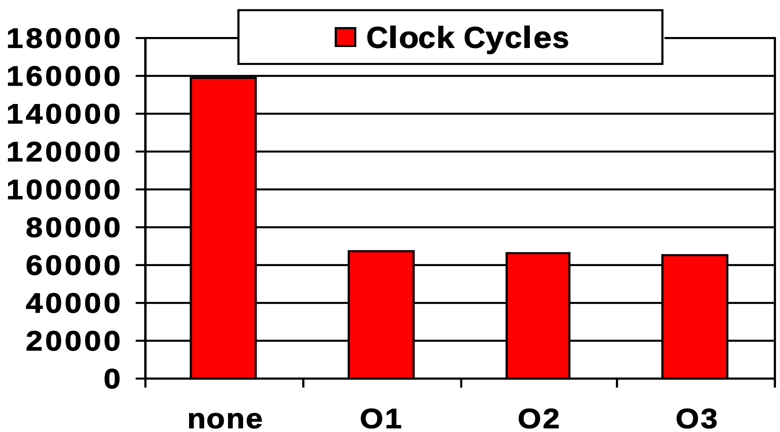
```
sort:     addi $sp,$sp, –20        # make room on stack for 5 registers
          sw $ra, 16($sp)          # save $ra on stack
          sw $s3,12($sp)           # save $s3 on stack
          sw $s2, 8($sp)           # save $s2 on stack
          sw $s1, 4($sp)           # save $s1 on stack
          sw $s0, 0($sp)           # save $s0 on stack
          …                        # procedure body
          …
exit1:    lw $s0, 0($sp)           # restore $s0 from stack
          lw $s1, 4($sp)           # restore $s1 from stack
          lw $s2, 8($sp)           # restore $s2 from stack
          lw $s3,12($sp)           # restore $s3 from stack
          lw $ra,16($sp)           # restore $ra from stack
          addi $sp,$sp, 20         # restore stack pointer
          jr $ra                   # return to calling routine
```

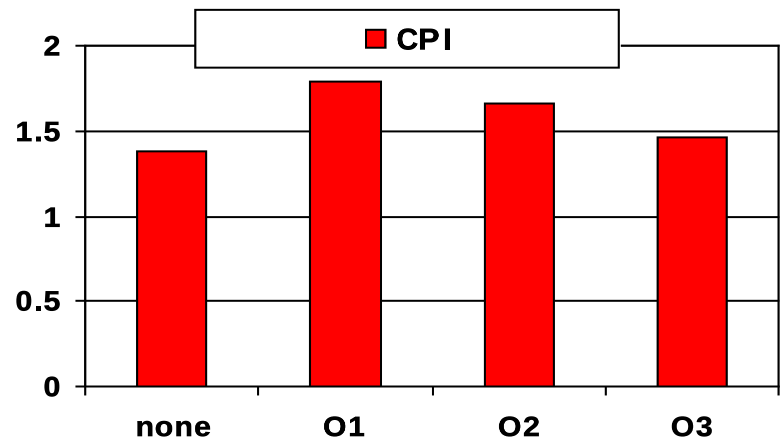
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

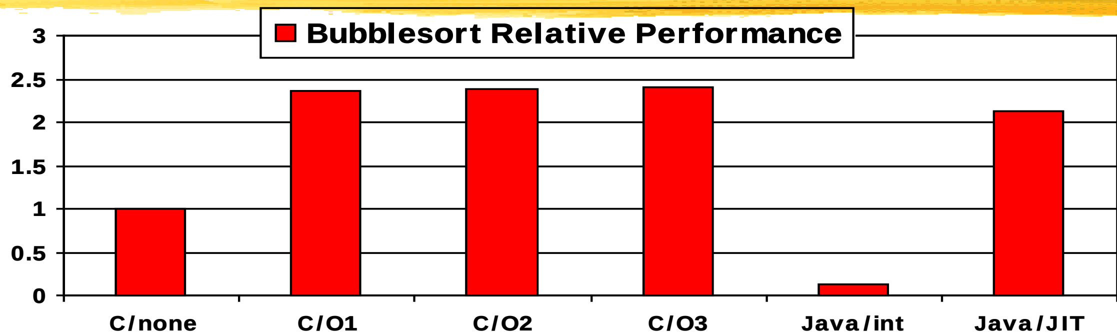


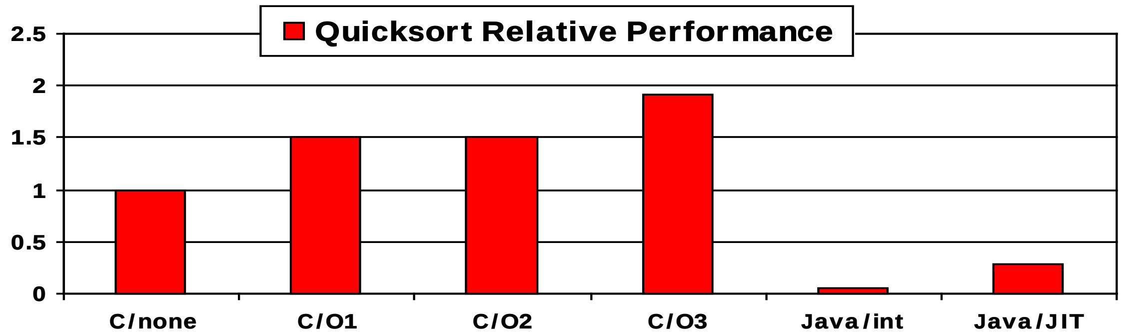


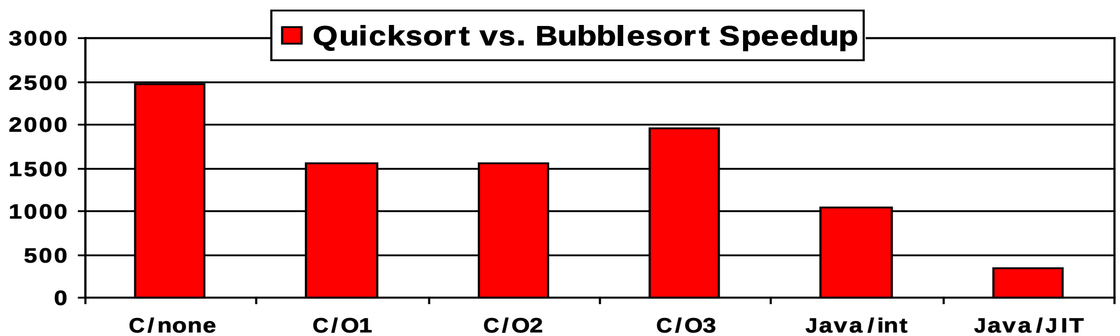




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Clearing Example - Array vs. Pointer

<pre>clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}</pre>
	<pre>clear2(int *array, int size) {
	  int *p;
	  for (p = &array[0]; p < &array[size];
	        p = p + 1)
	     *p = 0;
	}</pre>

<pre>   move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2   # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1</pre>
	<pre> move $t0, $a0 # p = & array[0]
	       sll $t1, $a1, 2 # $t1 = size * 4
	       add $t2,$a0,$t1 # $t2 =
	                       # &array[size]
	loop2: sw $zero,0($t0) # Memory[p] = 0
	       addi $t0,$t0,4 # p = p + 4
	       slt $t3,$t0,$t2 # $t3 =
	                       #(p<&array[size])
	       bne $t3,$zero,loop2 # if (…)
	                           # goto loop2</pre>

Comparison of Array vs. Pointer

- Multiply "strength reduced" to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Outline

- Instruction set architecture
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Signed and unsigned numbers
- Representing instructions
- Operations
  - Logical
  - Decision making and branches
- Supporting procedures in hardware
- Communicating with people
- Addressing for 32-bit immediate and addresses
- Synchronization
- Translating and starting a program
- A sort example
- Arrays versus pointers
- ARM and x86 instruction sets

ARM & MIPS Similarities

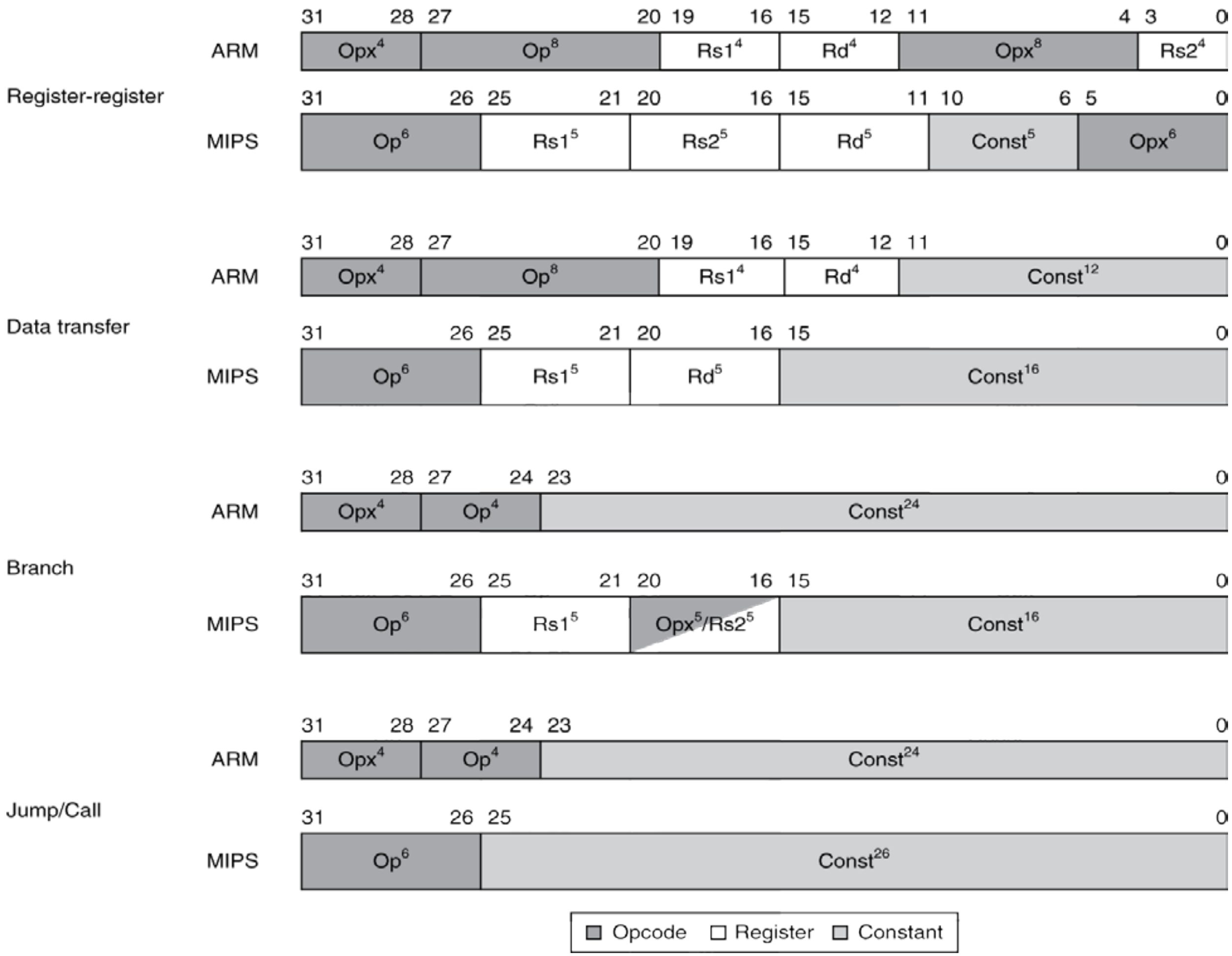
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 $\times$ 32-bit	31 $\times$ 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Encoding



IA-32 Overview

- **Complexity:**
  - **Instructions from 1 to 17 bytes long**
  - **one operand must act as both a source and destination**
  - **one operand can come from memory**
  - **complex addressing modes**
    ex: "base or scaled index with 8 or 32 bit displacement"
- **Saving grace:**
  - **the most frequently used instructions are not too difficult to build**
  - **compilers avoid the portions of the architecture that are slow**

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

The Intel x86 ISA

- **Evolution with backward compatibility**
  - **8080 (1974): 8-bit microprocessor**
    - **Accumulator, plus 3 index-register pairs**
  - **8086 (1978): 16-bit extension to 8080**
    - **Complex instruction set (CISC)**
  - **8087 (1980): floating-point coprocessor**
    - **Adds FP instructions and register stack**
  - **80286 (1982): 24-bit addresses, MMU**
    - **Segmented memory mapping and protection**
  - **80386 (1985): 32-bit extension (now IA-32)**
    - **Additional addressing modes and operations**
    - **Paged memory mapping as well as segments**

The Intel x86 ISA

- **Further evolution…**
  - **i486 (1989): pipelined, on-chip caches and FPU**
    - **Compatible competitors: AMD, Cyrix, …**
  - **Pentium (1993): superscalar, 64-bit datapath**
    - **Later versions added MMX (Multi-Media eXtension) instructions**
    - **The infamous FDIV bug**
  - **Pentium Pro (1995), Pentium II (1997)**
    - **New microarchitecture (see Colwell, *The Pentium Chronicles*)**
  - **Pentium III (1999)**
    - **Added SSE (Streaming SIMD Extensions) and associated registers**
  - **Pentium 4 (2001)**
    - **New microarchitecture**
    - **Added SSE2 instructions**

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers

Name	Use
EAX	GPR 0
ECX	GPR 1
EDX	GPR 2
EBX	GPR 3
ESP	GPR 4
EBP	GPR 5
ESI	GPR 6
EDI	GPR 7
CS	Code segment pointer
SS	Stack segment pointer (top of stack)
DS	Data segment pointer 0
ES	Data segment pointer 1
FS	Data segment pointer 2
GS	Data segment pointer 3
EIP	Instruction pointer (PC)
EFLAGS	Condition codes

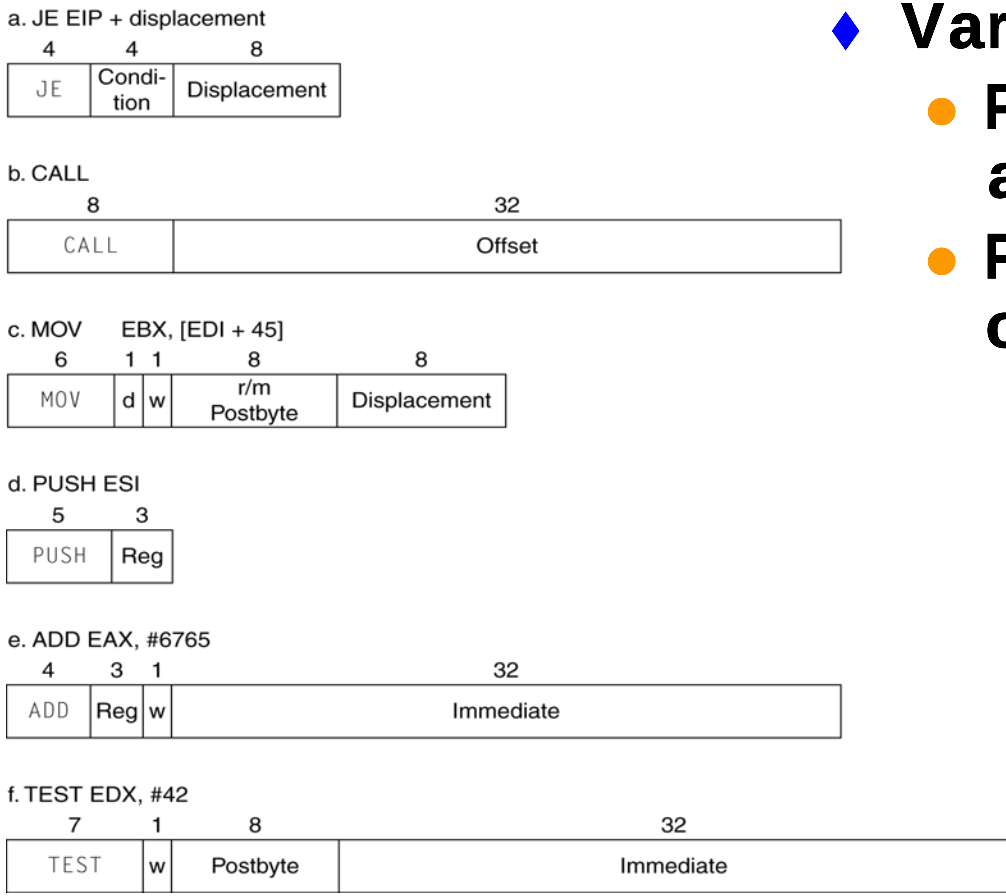
Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

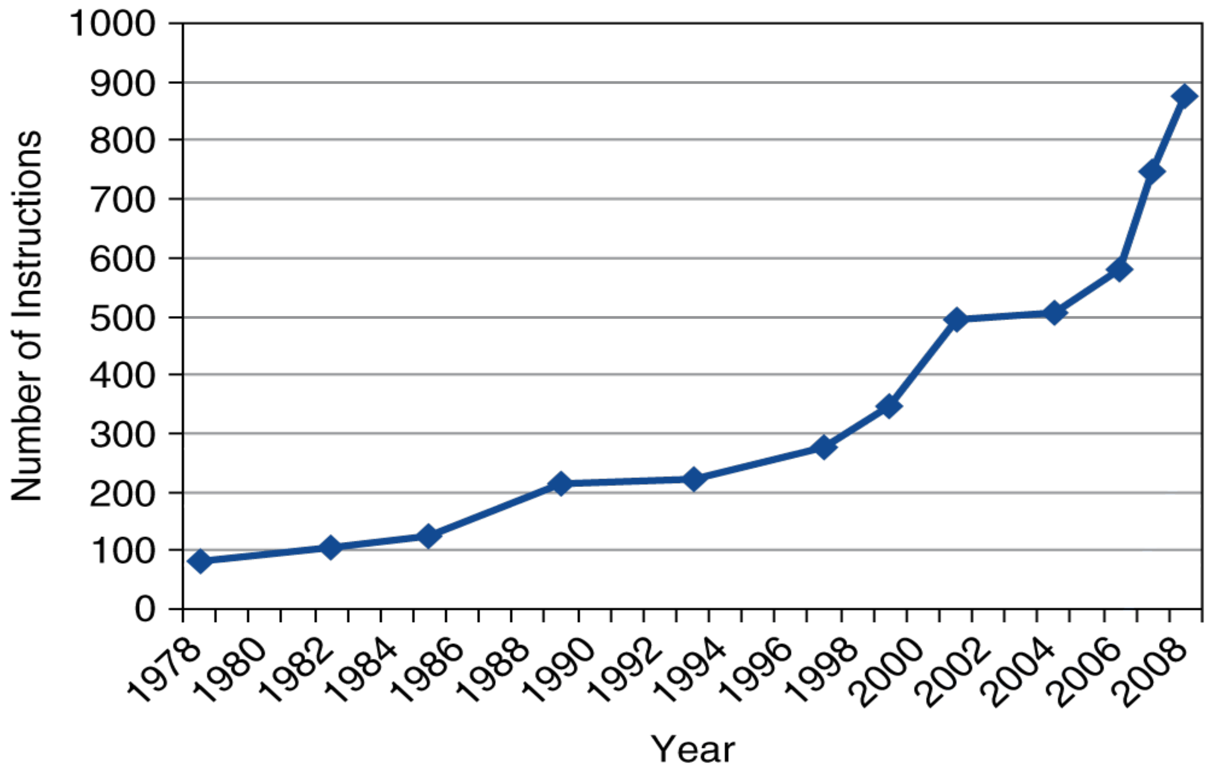
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity

Fallacies

- **Backward compatibility ⇒ instruction set doesn't change**
  - **But they do accrete more instructions**



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - Ex: passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - Compared to CISC ISA, ex: x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

MIPS Assembly Summary

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call