Chapter 2

Instructions

Instruction Set:

- **Language of the Computer**
- **Different computers have different instruction sets**
  - **But with many aspects in common**
- **Early computers had very simple instruction sets**
  - **Simplified implementation**
- **Many modern computers also have simple instruction sets**

The MIPS Instruction Set:

- We'll be working with the MIPS instruction set architecture
- MIPS (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture developed by MIPS Technologies (formerly MIPS Computer Systems, Inc.)
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo 64, Sony PlayStation, Cisco, Sun Micro Systems …
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Computer architecture courses in universities often study the MIPS architecture. (MIT, Stanford, Eastern, …)

MIPS arithmetic

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)

Example 1:

C/Java code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

MIPS arithmetic

- **Example 2:**

C/Java code:

```
A = B + C + D;
E = F - A;
```

MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

***Assume: registers $s1, $s2, $s3, $s4, and $s5 contain the <u>values</u> of B, C, D, E, and F, respectively**

MIPS arithmetic

- **Example 3:**

C/Java code:

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

MIPS code:

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

***Assume: registers $s0, $s1, $s2, $s3, and $s4 contain the <u>values</u> of f, g, h, i, and j, respectively**

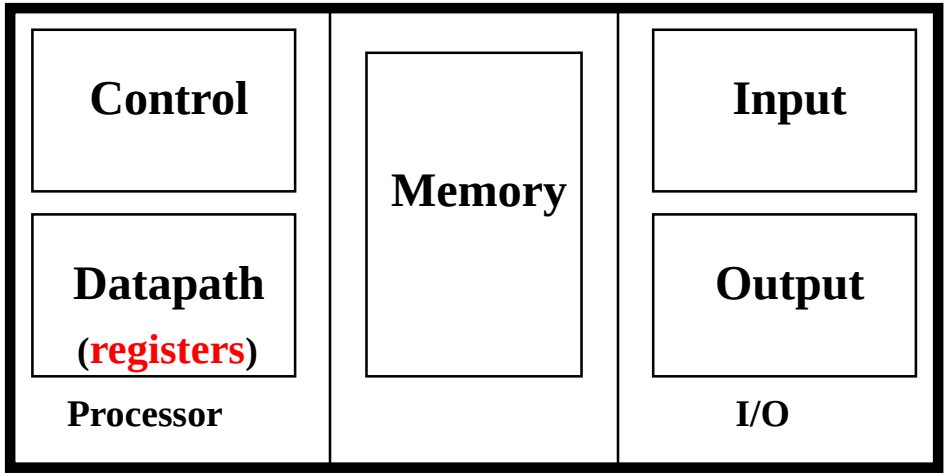
- **Operands must be registers, only 32 registers provided**
- ***Design Principle 2:* smaller is faster.**

Registers vs. Memory (1)

- **Arithmetic instructions operands must be registers,**
  - **Only 32 registers provided (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**
- **Compiler associates variables with registers**

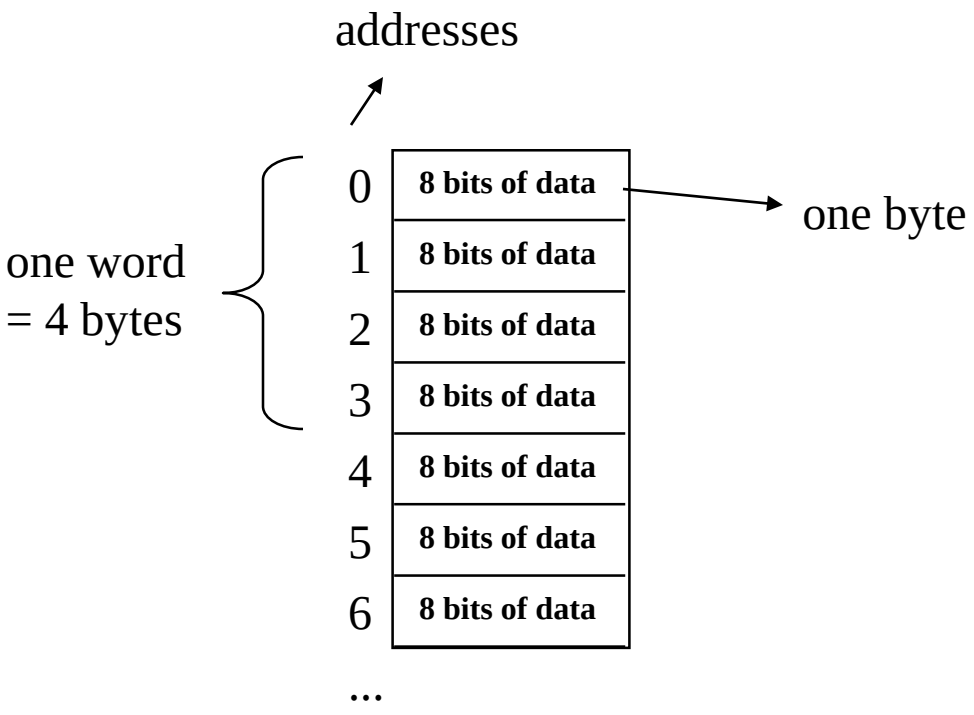
Registers vs. Memory (2)

- **Memory reference instructions:**
  - **lw (load word) : move data from memory to register**
  - **sw (store word) : move data from register to memory**



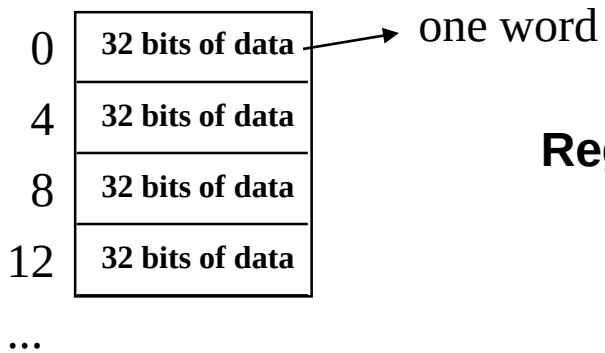
Memory Organization (1)

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



Memory Organization (2)

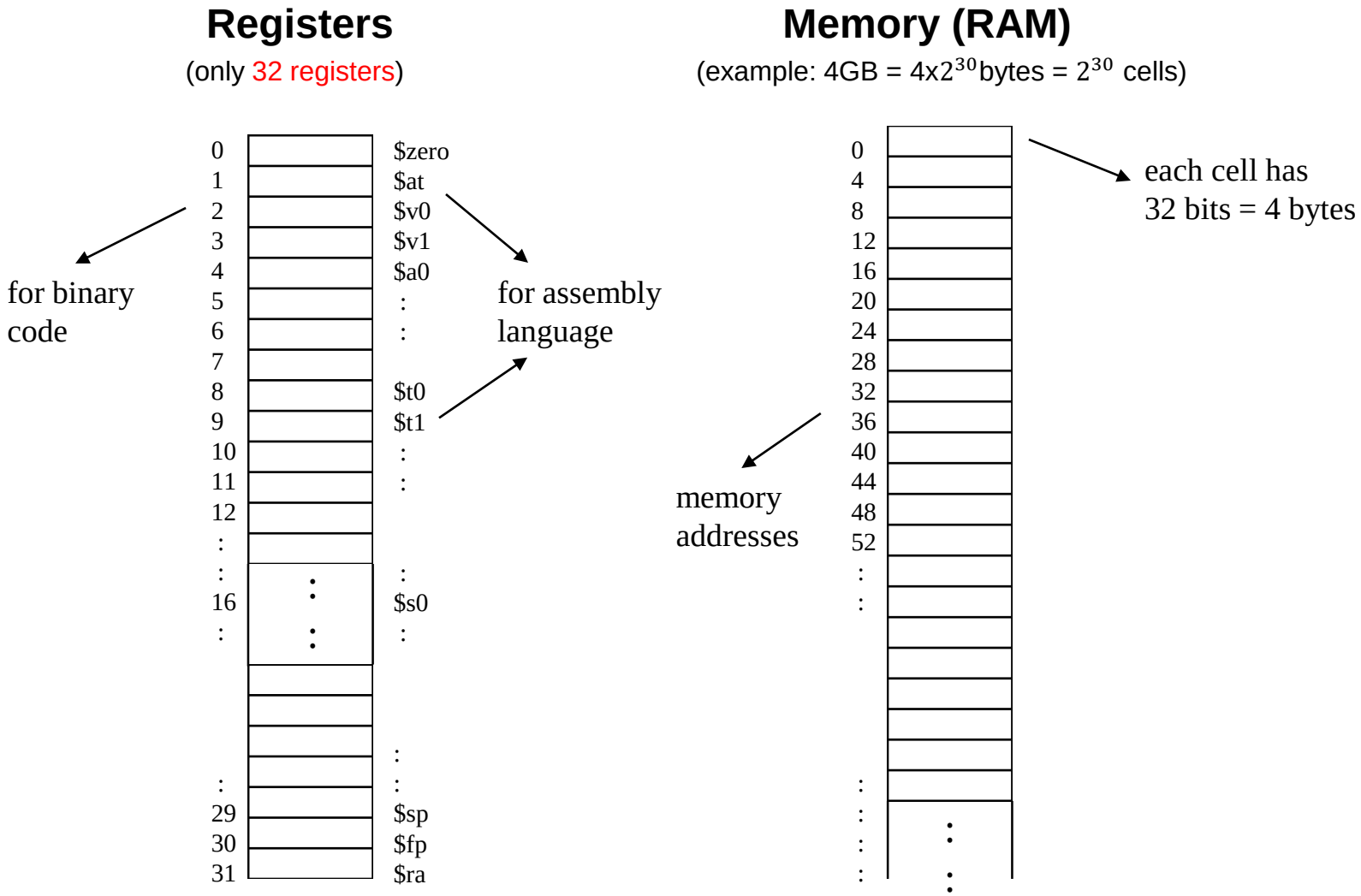
- **Bytes are nice, but most data items use larger "words"**
- **For MIPS, a word is 32 bits or 4 bytes.**



Registers hold 32 bits of data

- **4GB=4x2^{30} bytes=2^{32} bytes with byte addresses from 0,1,2, ...to 2^{32}-1**
- **4GB=2^{30} words with byte addresses 0, 4, 8, ... 2^{30}-1**
- **Main memory used for composite data (Arrays, structures …)**
- **To apply arithmetic operations**
  - **Load values from memory into registers (lw)**
  - **Store result from register to memory (sw)**

Registers vs. Memory (Summary)



Load and store Instructions

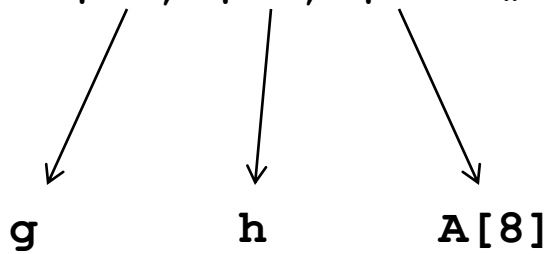
- **Example 1: (g in $s1, h in $s2, base address of A[ ] in $s3)**

C/Java code:

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

Compiled MIPS code:

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



Load and store Instructions

- **Example 2: (h in $s2, base address of A[ ] in $s3)**

C/Java code:

```
A[12] = h + A[8];
```

Compiled MIPS code:

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

- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory!**

More Example

- **Can we figure out the Java/C++ code?**

  Assume: $s4 contains address of v[0] and $s3 contains value k

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

```
swap:
    add $s1, $s3, $s3
    add $s2, $s1, $s1
    add $s2, $s4, $s2
    lw $s5, 0($s2)
    lw $s6, 4($s2)
    sw $s6, 0($s2)
    sw $s5, 4($s2)
    jr $ra
```

address of v[0]

k

Summary: 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!**

Immediate Operands (constants)

- **Constant data specified in an instruction**

```
addi $s3, $s3, 4
```

- **No subtract immediate instruction (no `subi`)**
  - **Just use a negative constant**

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

- **Load immediate**

```
li $s3, 4  (same as: addi $s3, $zero, 4)
```

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

The Constant Zero

- **MIPS register 0 (`$zero`) is the constant `0`**
  - **Cannot be overwritten**

- **Useful for common operations**
  - **E.g., move between registers**

```
add $t2, $s1, $zero
addi $s3, $zero, 4
```

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **<u>Instruction</u>** **<u>Meaning</u>**

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`
`addi $s1, $s2, 100`	`$s1 = $s2 + 100`
`addi $s1, $s2, -100`	`$s1 = $s2 – 100`
`add $s1, $s2, $zero`	`$s1 = $s2 + 0`
`li $s1, 4`	`$s1 = 4`

Numbers

- **Bits are just bits (no inherent meaning)**
  — conventions define relationship between bits and numbers
- **Binary numbers (base 2)**
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- **How do we represent negative numbers?**
  i.e., which bit patterns will represent which numbers?

Bases that are a power of two

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Converting to base 10

- The value of a specific number in a specified base is: $\sum_{i=0}^{n-1} d_i * b^i$,where d is a digit and b is the base.

- An example in base 10:
  $425_{10} = 4\times10^2 + 2\times10^1 + 5\times10^0 = 400 + 20 + 5$
- An example in base 2:
  $1011_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 8 + 0 + 2 + 1 = 11_{10}$
- An example in base 8:
  $425_8 = 4\times8^2 + 2\times8^1 + 5\times8^0 = 256 + 16 + 5 = 277_{10}$
- An example in base 16:
  $17A_{16} = 1\times16^2 + 7\times16^1 + 10\times16^0 = 256 + 112 + 10 = 378_{10}$

Base Conversion Algorithm

To convert from base 10 to base *m*:

Keep dividing the base 10 number by *m* until the quotient is 0, and save all of the remainders

Example: 25 (base 10) = 11001 (base 2)

25 div 2 = 12 … 1
12 div 2 = 6 … 0
6 div 2 = 3 … 0
3 div 2 = 1 … 1
1 div 2 = 0 … 1

***Read these numbers from bottom to top**

quotient remainder

Examples of Converting between Bases

- <u>Binary to Hexadecimal:</u>

  0100 1100 0011 1101

  => 4 C 3 D

  => 4C3D (base 16)

- <u>Binary to Octal:</u>

  011 110 001

  => 3 6 1

  => 361 (base 8)

- <u>Hexadecimal to Octal:</u>

  C3F0

  => 1100 0011 1111 0000

  => 00 1100 0011 1111 0000 (note: add 2 0s to the left)

  => 001 100 001 111 110 000

  => 1 4 1 7 6 0 (base 8)

Binary Representations

- **32 bit *unsigned* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
...
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Binary Representations - MIPS

- **Two's complement representation can represent both positive and negative values.**
- **32 bit *signed* numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — ***maxint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Operations

- **Negating a two's complement number:** ***<u>invert all bits and add 1</u>***
  - **remember: "negate" and "invert" are quite different!**

- **Example 1:**

$$3_{10} = 00000000000000000000000000000011_2$$

$$-3_{10} = 11111111111111111111111111111100_2 + 1$$

$$= 11111111111111111111111111111101_2$$

- **Example 2:**

$$-3_{10} = 11111111111111111111111111111101_2$$

$$3_{10} = 00000000000000000000000000000100_2 + 1$$

$$= 00000000000000000000000000000011_2$$

Two's Complement Operations

What are the decimal values of following 32-bit two's complement numbers?

- **Example 3:**

$$
\begin{aligned}
&11111111111111111111111111111100_2 \\
&= (1\times -2^{31}) + (1\times 2^{30}) + (1\times 2^{29}) + ... + (1\times 2^{2}) + (0\times 2^{1}) + (0\times 2^{0}) \\
&= (-2^{31} + 2^{30} + 2^{29} + ... + 2^{2} + 0 + 0)_{10} \\
&= -4_{10}
\end{aligned}
$$

- **Example 4:**

$$
\begin{aligned}
&11111111111111111111111111111101_2 \\
&= (1\times -2^{31}) + (1\times 2^{30}) + (1\times 2^{29}) + ... + (1\times 2^{2}) + (0\times 2^{1}) + (1\times 2^{0}) \\
&= (-2^{31} + 2^{30} + 2^{29} + ... + 2^{2} + 0 + 1)_{10} \\
&= -3_{10}
\end{aligned}
$$

Signed Extension Shortcut

- An integer register on the MIPS is 32 bits. When a value is loaded from memory with fewer than 32 bits, the remaining bits must be assigned.
- Example: convert 16-bit binary number of 2 into 32-bit binary numbers 2 and -2:

$$0000000000000010_2 = 2_{10}$$

$$+2: 00000000000000000000000000000010_2$$

$$-2: 11111111111111111111111111111110_2$$

$$Note:$$

$$1111111111111101_2 + 1_2 = 1111111111111110_2$$

$$then\ copying\ the\ sign\ bit\ 16\ times\ and\ placing\ it\ on\ the\ left$$

$$11111111111111111111111111111110_2 = -2_{10}$$

Binary addition

		(0)	(0)	(1)	(1)	(0)	(0)	→ (*carries*)
	….	0	0	0	1	1	1	
+	….	0	0	0	1	1	0	
	….	0	0	1	1	0	1	

		(1)	(1)	(1)	(1)	(0)	(0)	→ (*carries*)
	….	0	0	1	0	1	0	
+	….	0	1	1	1	1	0	
	….	1	0	1	0	0	0	

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)

Machine Language (R-format: for register)

- **Instructions, like registers and words of data, are also 32 bits long**
  - **Example:** `add $t0, $s1, $s2`
  - **registers have numbers,** `$t0=8, $s1=17, $s2=18`

- **Instruction Format:**

000000	10001	10010	01000	00000	100000

op	rs	rt	rd	shamt	funct

- ***op(opcode): basic operation of the instruction***
- ***rs: the first register***
- ***rt: the second register***
- ***rd: the destination register***
- ***shamt: sfift amount***
- ***funct: function code***

Machine Language (R-format)

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

```
add $t0, $s1, $s2     # $t0=8, $s1=17, $s2=18
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000100011001001000000001000000_2 = 02324020_{16}$

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 -- allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Machine Language (I-format: data transfer)

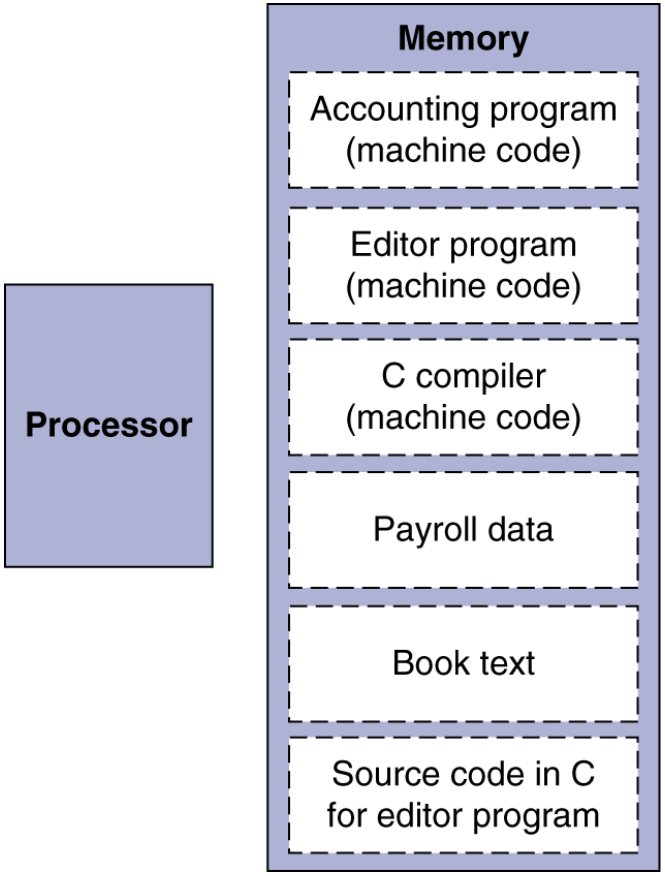
- **Introduce a new type of instruction format**
  - **I-type for data transfer instructions**
  - **other format was R-type for register**
- **Example:** `lw $t0, 32($s2)`

35	**18**	**8**	**32**
op	**rs**	**rt**	**16 bit number (address)**

Stored Program Concept

- Instructions (programs) represented in binary, just like data
- Instructions are stored in memory -- to be read or written just like data
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

The BIG Picture



Logical Operators

	C/C++	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>	`srl`
Logical and	&	&	`and, andi`
Logical or	|	|	`or, ori`
Logical Not	~	~	`nor`

<u>Example:</u>

```
sll $t2, $s0, 8
```

// left shift $s0 8 bits and store the result to $t2
// Note: the content of $s0 is unchanged

Shift Operations

- **Use R-format**

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**
  - **Note: sll by *i* bits = multiplies by 2^i**
- **Shift right logical**
  - **Shift right and fill with 0 bits**
  - **Note: srl by *i* bits = divides by 2^i (unsigned only)**

Logical Operations - sll

- **Shift left logical (sll):**

Instruction	**Meaning**
`sll $t2, $s0, 8`	`$t2 ← $s0 << 8 bits`

R-Format

op	rs	rt	rd	shamt	funct	
0	0	16	10	8	0	(decimal)
000000	00000	10000	01010	01000	000000	(binary)

unused

always 0

Example: **$s0 contains: 0000 0000 0000 0000 0000 0000 0000 1101**

after "`sll $t2, $s0, 8`"

$t2 contains: 0000 0000 0000 0000 0000 1101 0000 0000

Fill empty bits with 0s

Logical Operations - srl

- **Right left logical (srl):**

Instruction	**Meaning**
`srl $t2, $s0, 8`	`$t2 ← $s0 >> 8 bits`

Example:

$s0 contains: 0000 0000 0000 0000 0000 0000 0000 1101

after "`srl $t2, $s0, 8`"

$t2 contains: 0000 0000 0000 0000 0000 0000 0000 0000

Fill empty bits with 0s

Logical Operations - and

Instruction	Meaning
`and $t0, $t1, $t2`	`$t0 ← $t1 and $t2`

Example:

$t1 contains: 0000 0000 0000 0000 0000 0000 0010 1101

$t2 contains: 0000 0000 0000 0000 0000 0000 0011 0101

after "and $t0, $t1, $t2"

$t0 contains: 0000 0000 0000 0000 0000 0000 0010 0101

Logical Operations - or

Instruction	Meaning
`or $t0, $t1, $t2`	`$t0 ← $t1 or $t2`

Example:

$t1 contains: 0000 0000 0000 0000 0000 0000 0010 1101

$t2 contains: 0000 0000 0000 0000 0000 0000 0011 0101

after "or $t0, $t1, $t2"

$t0 contains: 0000 0000 0000 0000 0000 0000 0011 1101

Logical Operations - NOT

- **Useful to invert bits in a word: Change 0 to 1, and 1 to 0**
- **Note: NOT (*a* OR *b*) ≡ *a* NOR *b***
- **MIPS has NOR instruction only**

  So, Not *a* ≡ NOT (*a* OR *0*) ≡ *a* NOR *0*

Instruction	**Meaning**
`nor $t0, $t1, $zero`	`$t0 ← NOT ($t1 or $zero)`
	`$t0 ← NOT $t1`

Example:

$t1 contains: 0000 0000 0000 0000 0000 0000 0010 1101

$zero contains: 0000 0000 0000 0000 0000 0000 0000 0000

after "nor $t0, $t1, $zero"

$t0 contains: 1111 1111 1111 1111 1111 1111 1101 0010

Control (if statement)

- **Decision making instructions**
  - **alter the control flow,**
  - **i.e., change the "next" instruction to be executed**

- **MIPS conditional branch instructions (bne, beq: I-format):**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

→ **if ($t0 != $t1) go to Label**
→ **if ($t0 == $t1) go to Label**

- **Example:** `if (i==j) h = i + j;`

-

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

h i j

Control (if..else.. Statement)

- **MIPS unconditional branch instructions:**

```
j   label
```

- **J-format:**

op	26 bit address

- **Example:**

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

```
                i     j        h
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1:  sub $s3, $s4, $s5
Lab2:  ...
```

Control (while loops)

- **C/Java code (assuming *i* in $s3, *k* in $s5, address of A in $s6):**

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

- **Compiled MIPS code:**

i

```
Loop: sll  $t1, $s3, 2      # $t1 ← 4 * i
      add  $t1, $t1, $s6    # $t1 ← address A[i]
      lw   $t0, 0($t1)      # $t0 ← value A[i]
      bne  $t0, $s5, Exit   # if (A[i] != k)
      addi $s3, $s3, 1      # else i = i+1
      j    Loop             # go to Loop
Exit: …
```

So far:

- **Instruction** **Meaning**

```
add $s1,$s2,$s3    $s1 = $s2 + $s3
sub $s1,$s2,$s3    $s1 = $s2 - $s3
lw $s1,100($s2)    $s1 = Memory[$s2+100]
sw $s1,100($s2)    Memory[$s2+100] = $s1
bne $s4,$s5,L      Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L      Next instr. is at Label if $s4 = $s5
j Label            Next instr. is at Label
```

- **Formats:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Control Flow (slt, slti)

- **We have: beq, bne, what about Branch-if-less-than?**
- **New instructions:**

MIPS:	**Meaning:**

`slt $t0, $s1, $s2`
(R-format)
	`if ($s1 < $s2) then`
	`  $t0 = 1`
	`else`
	`  $t0 = 0`

`slti $t0, $s1, 10`
(I-format)
	`if ($s1 < 10) then`
	`  $t0 = 1`
	`else`
	`  $t0 = 0`

Control Flow (`slt`, `slti`)

- `slt` and `slti` are used in combination with `beq` and `bne`

- **Example 1:**

```
slt $t0, $s1, $s2          # if ($s1 < $s2)
bne $t0, $zero, Label      # branch to Label, skip {do ...}
   {do something ...}         same as
 Label:                    # if ($s1 >= $s2)
                           # {do something ...}
```

- **Example 2:**

```
slti $t0, $s1, 100         # if ($s1 >= 100)
beq $t0, $zero, Label      # branch to Label, skip {do...}
   {do something ...}         same as
 Label:                    # if ($s1 < 100)
                           # {do something ...}
```

Control Flow (`slt`, `slti`)

Example 3:

Java/C++:

```
if (x < y)
   { label1: do something ...}
else
   { label2: do something ...}
```

MIPS:

```
                x     y
        slt $t0, $s0, $s1
        bne $t0, $zero, Label1   # if x < y, go to Label1
        j Label2                 # else go to Label2
Label1: {do something ...}
        j Exit
Label2: {do something ...}
  Exit:
```

Control (for loops)

- C/Java code (assuming *i* in $s3, *k* in $s5, address of A in $s6):

```
for (i=0; i<k; i++)
    A[i]=0;
```

- Compiled MIPS code:

```
                                i   k

      li    $S3, 0                  # i=0 (addi $s3, $zero, 0)
Loop: slt   $t0, $s3, $s5           # test if i<k
      beq   $t0, $zero, Exit        # if i>=k, go to Exit
      sll   $t1, $s3, 2             # $t1 ← (4*i)
      add   $t1, $t1, $s6           # $t1 ← address A[i]
      sw    $zero, 0($t1)           # A[i]← 0
      addi  $s3, $s3, 1             # i = i+1
      j     Loop                    # go to Loop
Exit: …

                                     A[]
```

Branch Instruction Design

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

Signed vs. Unsigned

- **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`

Register Usage

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B - 1;
  C = 100;

- Solutions
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:

```
addi $s2, $s2, 5      → $s2 = $s2 + 5
slti $s1, $s2, 10     → if ($s2 < 10) then $s1 = 1 else $s1 = 0
addi $s2, $s2, -6     → $s2 = $2 – 6
li $s1, 100           → $s1 = 100
```

- Most constants are small → 16-bit immediate is sufficient
- For the occasional 32-bit constant → see next slide

How about larger constants (32-bit)?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 0000101010101010
```

filled with zeros

0000101010101010	0000000000000000

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 0000000000000111
```

0000101010101010	0000000000000000
0000000000000000	0000000000000111

ori

0000101010101010	0000000000000111

this is the value we need

Branch Addressing

- **Branch instructions specify**
  - **Opcode, two registers, target address**
- **Most branch targets are near branch**
  - **Forward or backward**

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

- **PC-relative addressing**
  - **Target address = PC + offset × 4**
  - **PC already incremented by 4 by this time**

Program Counter

Jump Addressing

- **Jump (`j` and `jal`) targets could be anywhere in text segment**
  - **Encode full address in instruction**

op	address
6 bits	26 bits

- **(Pseudo)Direct jump addressing**
  - **Target address = PC : (address × 4)**

Program Counter

Target Addressing Example

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

20000*4=80000 (for jump)

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

PC (80016)+2*4=80024

Assembler Pseudoinstructions

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

MIPS:		Meaning:
`move $t0, $t1`	→	`add $t0, $zero, $t1`
`li $t0, 100`	→	`addi $t0, $zero, 100`
`blt $t0, $t1, L`	→	`slt $at, $t0, $t1` `bne $at, $zero, L`

Note: $at (register 1): assembler temporary

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

Procedure Calling Instructions

- **Procedure call: jump and link**

```
jal ProcedureLabel
```

  - **Address of following instruction put in `$ra`**
  - **Jumps to target address**

- **Procedure return: jump register**

```
jr $ra
```

  - **Copies `$ra` to program counter**
  - **Can also be used for computed jumps**
    - e.g., for case/switch statements

Procedure Test

- **C/C++ code:**

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

  – Arguments g, h, i, j in $a0, $a1, $a2, $a3
  – f in $s0 (hence, need to save $s0 on stack $sp)
  – Result in $v0

Procedure Example

- **MIPS code:**

```
Test:
    addi $sp, $sp, -8
    sw   $s0, 0($sp)
    sw   $ra, 4($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $ra, 4($sp)
    lw   $s0, 0($sp)
    addi $sp, $sp, 8
    jr   $ra
```

Procedure Label

Adjust stack pointer
Save $ra, $s0 on stack

Procedure body
f = (g + h) - (i + j);

Result

Restore $ra and $s0
Adjust stack pointer

Return

C/C++ Sort Example

- **Illustrates use of assembly instructions for a C/C++ bubble sort function**

- **Swap procedure to swap v[k] and v[k+1]:**

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

  – ***v*** **in** ***$a0***, ***k*** **in** ***$a1***, ***temp*** **in** ***$t0***

The MIPS Procedure Swap

```
swap:
      addi $sp, $sp, -4 # Adjust stack pointer
      sw   $ra, 0($sp)  # save return address
      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)
      lw   $ra, 0($sp)  # Adjust stack pointer
      addi $sp, $sp, 4  # restore return address
      jr $ra            # return to calling
                        #  routine
```

The Sort Procedure in C/C++

- **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); // to swap v[k] and v[k+1]
    }
  }
}
```

  – ***v*** **in** ***$a0***, ***k*** **in** ***$a1***, ***i*** **in** ***$s0***, ***j*** **in** ***$s1***

The MIPS 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 MIPS 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
```

add the MIPS procedure body and swap procedure here

Lessons Learned

- 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!

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Overview of MIPS

- **simple instructions all 32 bits wide**
- **very structured, no unnecessary baggage**
- **only three instruction formats**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- ***op(opcode): basic operation of the instruction***
- ***rs: the first register***
- ***rt: the second register***
- ***rd: the destination register***
- ***shamt: sfift amount***
- ***funct: function code***

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` $\neq$ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4` = `$t5`
`j Label`	Next instruction is at Label

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Addresses are not 32 bits**
  - **How do we handle this with load and store instructions?**

To summarize:

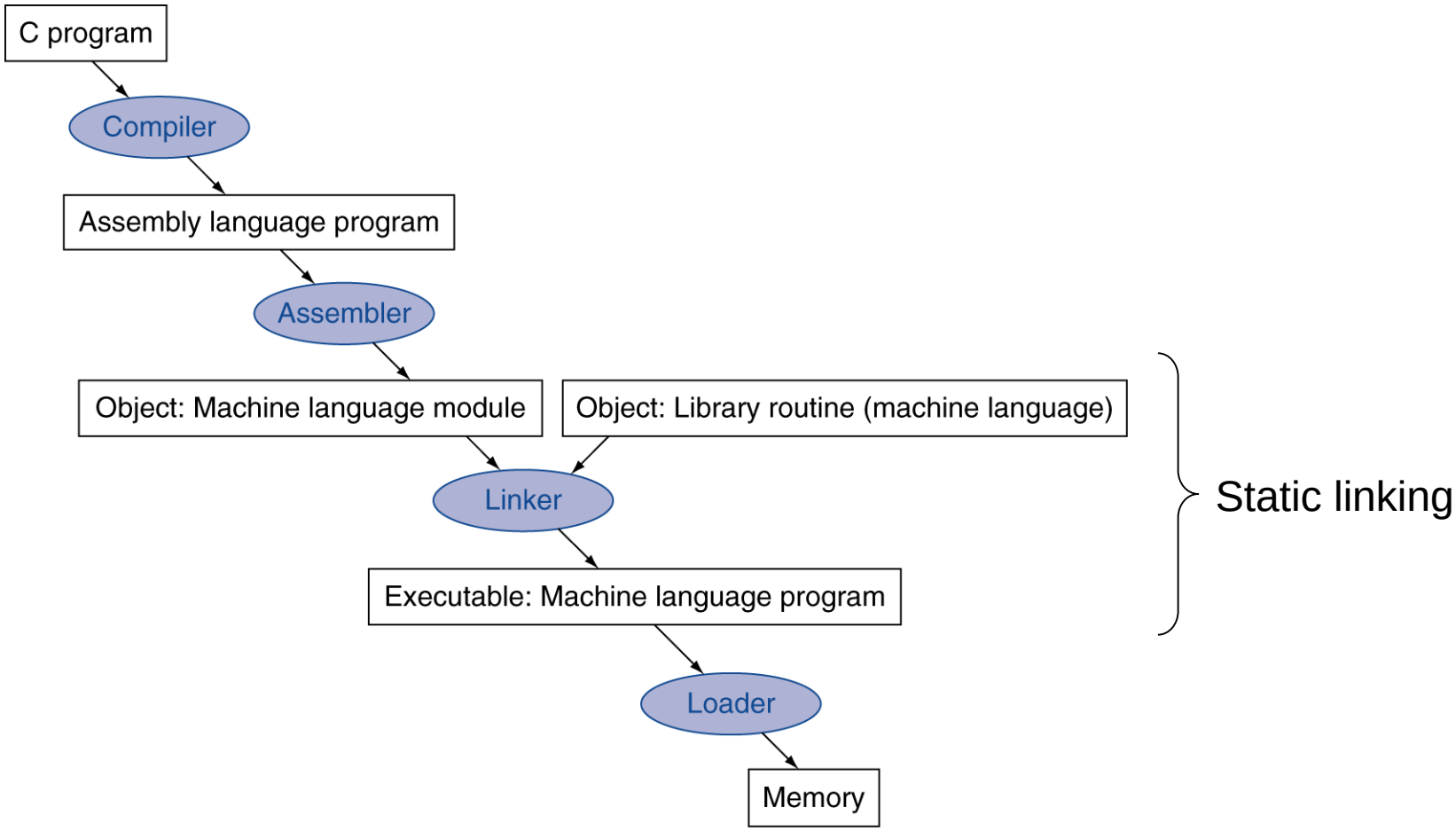
MIPS operands

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
Uncondi-tional 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

Translation and Starting a Program:



Fallacies

- Fallacy: More powerful instructions mean higher performance.
- Fallacy: Write in assembly language to obtain the highest performance.

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!