

CS/ECE 3810: Computer Organization

Lecture 4: MIPS instruction set

Anton Burtsev
August, 2022

Instruction Set

- Understanding the language of the hardware is key to understanding the hardware/software interface
- A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently
- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)
- What are important design principles when defining the instruction set architecture (ISA)?

Instruction Set

- Important design principles when defining the instruction set architecture (ISA):
 - keep the hardware simple – the chip must only implement basic primitives and run fast
 - keep the instructions regular – simplifies the decoding/scheduling of instructions

We will later discuss RISC vs CISC

A Basic MIPS Instruction

C code: $a = b + c ;$

Assembly code: (human-friendly machine instructions)

`add a, b, c # a is the sum of b and c`

A Basic MIPS Instruction

C code: $a = b + c ;$

Assembly code: (human-friendly machine instructions)
`add a, b, c # a is the sum of b and c`

Machine code: (hardware-friendly machine instructions)
`00000010001100100100000000100000`

Translate the following C code into assembly code:
 $a = b + c + d + e ;$

Example

C code `a = b + c + d + e;`
translates into the following assembly code:

Example

C code `a = b + c + d + e;`
translates into the following assembly code:

<code>add a, b, c</code>		<code>add a, b, c</code>
<code>add a, a, d</code>	or	<code>add f, d, e</code>
<code>add a, a, e</code>		<code>add a, a, f</code>

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others... the second sequence needs one more (temporary) variable `f`

Subtract Example

C code $f = (g + h) - (i + j);$

Assembly code translation with only add and sub instructions:

Subtract Example

C code $f = (g + h) - (i + j);$
translates into the following assembly code:

add t0, g, h		add f, g, h
add t1, i, j	or	sub f, f, i
sub f, t0, t1		sub f, f, j

Operands

- In C, each “variable” is a location in memory
- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers
- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed... there can be only so many scratchpad registers

Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32 bits wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word
- To make the code more readable, registers are partitioned as \$s0-\$s7 (C/Java variables), \$t0-\$t9 (temporary variables)...

Binary Stuff

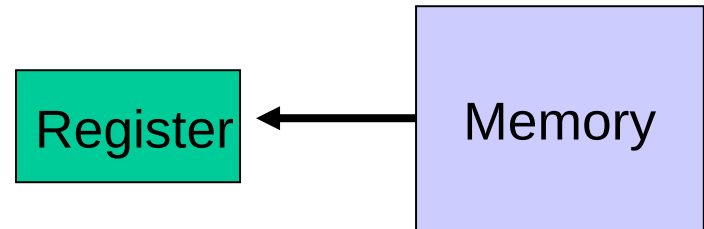
- 8 bits = 1 Byte, also written as 8b = 1B
- 1 word = 32 bits = 4B
- 1KB = 1024 B = 2^{10} B
- 1MB = 1024 x 1024 B = 2^{20} B
- 1GB = 1024 x 1024 x 1024 B = 2^{30} B
- A 32-bit memory address refers to a number between 0 and $2^{32} - 1$, i.e., it identifies a byte in a 4GB memory

Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

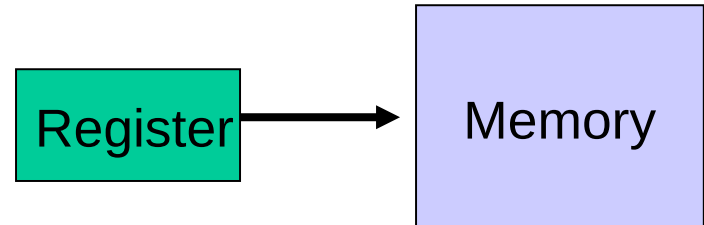
Load word

`lw $t0, memory-address`



Store word

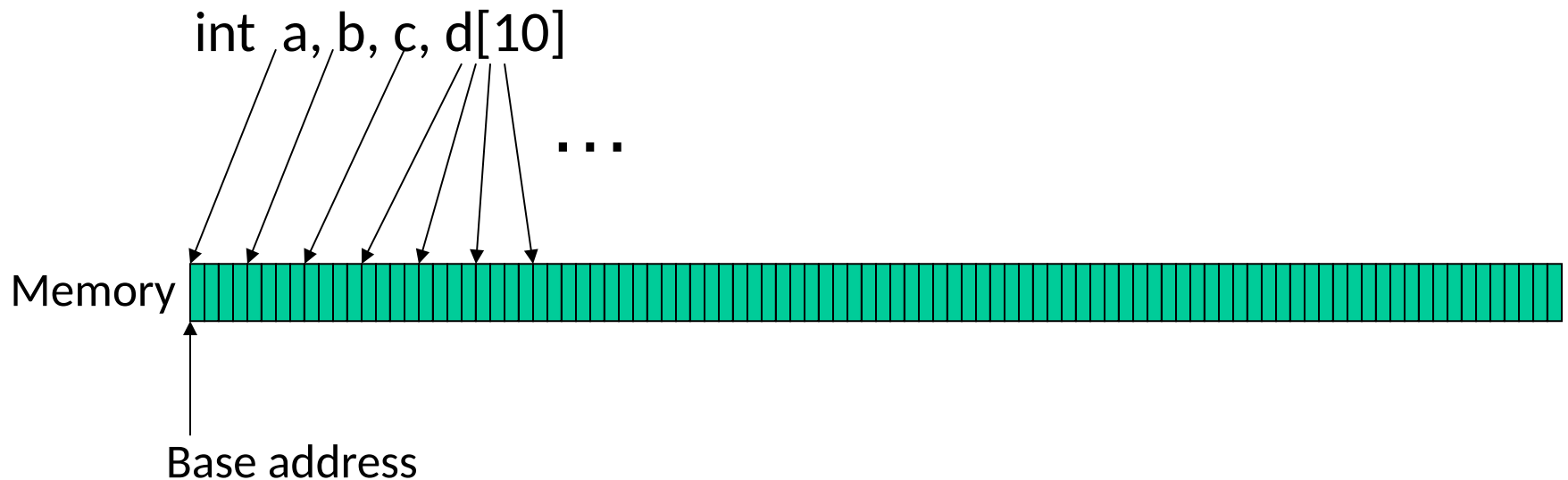
`sw $t0, memory-address`



How is memory-address determined?

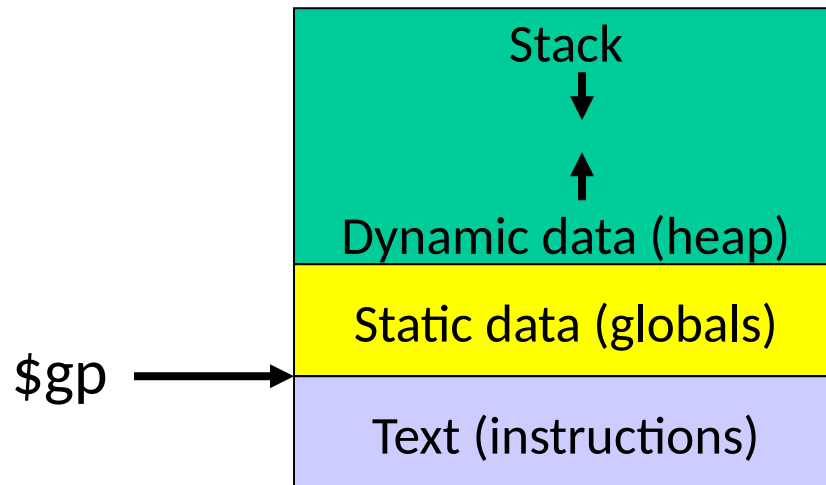
Memory Address

- The compiler organizes data in memory... it knows the location of every variable (saved in a table)... it can fill in the appropriate mem-address for load-store instructions



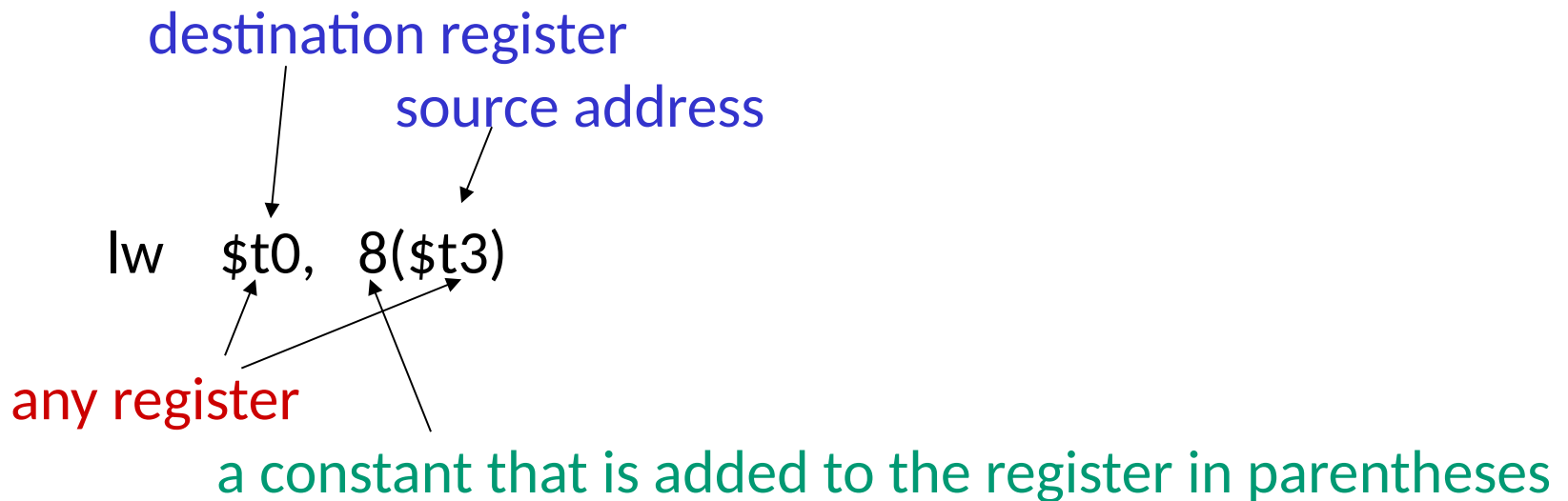
Memory Organization

\$gp points to area in memory that saves global variables



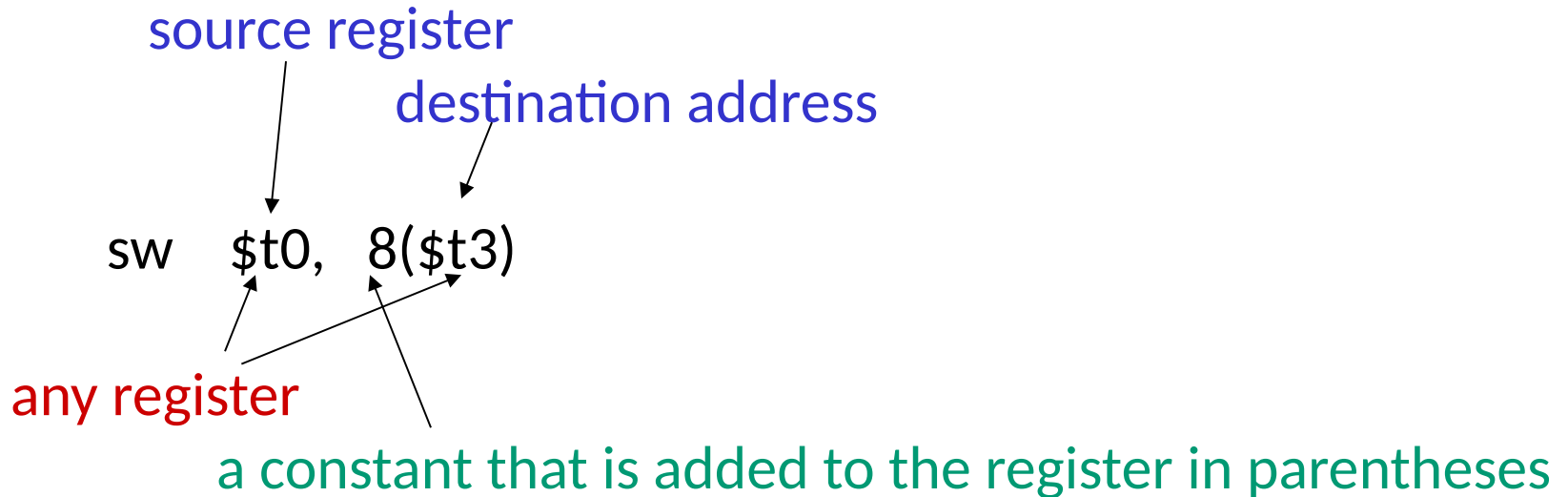
Memory Instruction Format

- The format of a load instruction:



Memory Instruction Format

- The format of a store instruction:



Example

```
int a, b, c, d[10];
```

```
addi $gp, $zero, 1000 # assume that data is stored at  
                        # base address 1000; placed in $gp;  
                        # $zero is a register that always  
                        # equals zero
```

```
lw  $s1, 0($gp) # brings value of a into register $s1  
lw  $s2, 4($gp) # brings value of b into register $s2  
lw  $s3, 8($gp) # brings value of c into register $s3  
lw  $s4, 12($gp) # brings value of d[0] into register $s4  
lw  $s5, 16($gp) # brings value of d[1] into register $s5
```

Example

Convert to assembly:

Remember: `int a, b, c, d[10];`

C code: `d[3] = d[2] + a;`

Example

Convert to assembly:

Remember: `int a, b, c, d[10];`

C code: `d[3] = d[2] + a;`

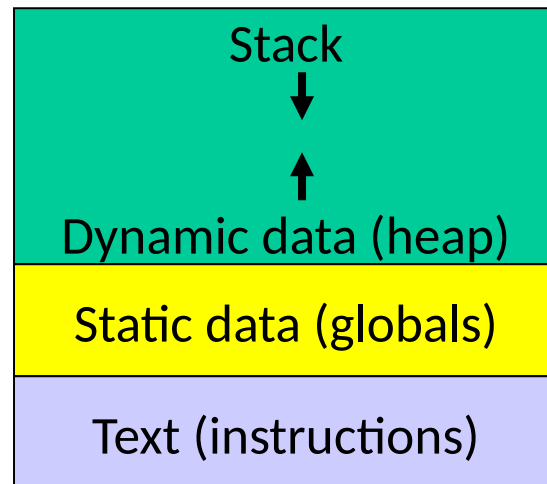
Assembly (same assumptions as previous example):

```
lw    $s0, 0($gp)    # a is brought into $s0
lw    $s1, 20($gp)   # d[2] is brought into $s1
add   $s2, $s0, $s1  # the sum is in $s2
sw    $s2, 24($gp)   # $s2 is stored into d[3]
```

Assembly version of the code continues to expand!

Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to \$fp as \$sp may change during the execution of the procedure
- \$gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



Binary Representation

- The binary number

01011000 00010101 00101110 11100111

Most significant bit ← ← Least significant bit

represents the quantity

$$0 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \dots + 1 \times 2^0$$

- A 32-bit word can represent 2^{32} numbers between 0 and $2^{32}-1$
... this is known as the unsigned representation as we're assuming that numbers are always positive

Negative Numbers

32 bits can only represent 2^{32} numbers – if we wish to also represent negative numbers, we can represent 2^{31} positive numbers (incl zero) and 2^{31} negative numbers

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = 2^{31}-1$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = -2^{31}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = -(2^{31} - 1)$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = -(2^{31} - 2)$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = -1$$

2's Complement

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}

...

0111 1111 1111 1111 1111 1111 1111 1111_{two} = $2^{31}-1$

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2^{31}

1000 0000 0000 0000 0000 0000 0000 0001_{two} = $-(2^{31} - 1)$

1000 0000 0000 0000 0000 0000 0000 0010_{two} = $-(2^{31} - 2)$

...

1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2

1111 1111 1111 1111 1111 1111 1111 1111_{two} = -1

Why is this representation favorable?

Consider the sum of 1 and -2 we get -1

Consider the sum of 2 and -1 we get +1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$$x_{31} - 2^{31} + x_{30} 2^{30} + x_{29} 2^{29} + \dots + x_1 2^1 + x_0 2^0$$

2's Complement

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}
0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}
...
0111 1111 1111 1111 1111 1111 1111 1111_{two} = $2^{31}-1$

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2^{31}
1000 0000 0000 0000 0000 0000 0000 0001_{two} = $-(2^{31} - 1)$
1000 0000 0000 0000 0000 0000 0000 0010_{two} = $-(2^{31} - 2)$
...
1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2
1111 1111 1111 1111 1111 1111 1111 1111_{two} = -1

Note that the sum of a number x and its inverted representation x' always equals a string of 1s (-1).

$$x + x' = -1$$

$x' + 1 = -x$... hence, can compute the negative of a number by

$-x = x' + 1$ inverting all bits and adding 1

Similarly, the sum of x and $-x$ gives us all zeroes, with a carry of 1

In reality, $x + (-x) = 2^n$... hence the name 2's complement

Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:
5, -5, -6

Example

- Compute the 32-bit 2's complement representations for the following decimal numbers:
5, -5, -6

5: 0000 0000 0000 0000 0000 0000 0000 0101

-5: 1111 1111 1111 1111 1111 1111 1111 1011

-6: 1111 1111 1111 1111 1111 1111 1111 1010

Given -5, verify that negating and adding 1 yields the number 5

Recap – Numeric Representations

- Decimal $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)
 $0x23$ or $23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0$

0-15 (decimal)  0-9, a-f (hex)

Dec	Binary	Hex
0	0000	00
1	0001	01
2	0010	02
3	0011	03

Dec	Binary	Hex
4	0100	04
5	0101	05
6	0110	06
7	0111	07

Dec	Binary	Hex
8	1000	08
9	1001	09
10	1010	0a
11	1011	0b

Dec	Binary	Hex
12	1100	0c
13	1101	0d
14	1110	0e
15	1111	0f

Constant or Immediate Operands

- We often use constants in operations

- Example: add 4 to register \$s3

```
lw $t0, AddrConstant4($s1) # $t0 = constant 4
```

```
add $s3,$s3,$t0 # $s3 = $s3 + $t0 ($t0 == 4)
```

- A more elegant way

```
addi $s3,$s3,4 # $s3 = $s3 + 4
```

Instruction format (R-type)

- Instructions are 32bit words in memory

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

- *op*: Basic operation of the instruction, traditionally called the opcode
- *rs*: The first register source operand
- *rt*: The second register source operand
- *rd*: The register destination operand. It gets the result of the operation
- *shamt*: Shift amount
- *funct*: Function/function code, selects the specific variant of the operation in the op field

Instruction format (R-type)

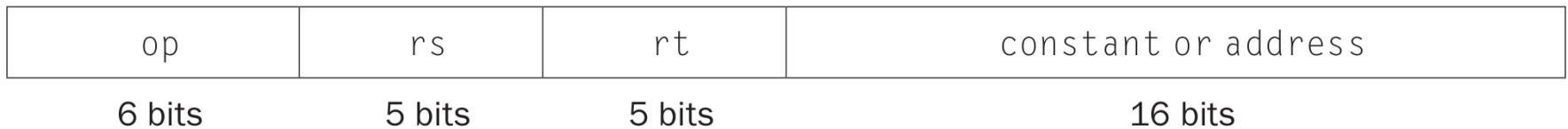
- Instructions are 32bit words in memory

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

- This works ok well for instructions like
 - `add $s0, $s1, $s3`
- But what about
 - `lw $t0, 32($s0)`
 - `addi $t0, $t1, 4 # t0 = t1 + 4`

Instruction format (I-type)

- Instructions are 32bit words in memory



- *op*: Basic operation of the instruction, traditionally called the opcode
- *rs*: The first register source operand
- *rt*: New meaning – destination register

Examples

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immediate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

Register numbers

- `$s0 - $s7` map on hardware registers
16 – 23
 - E.g., `$s0` is 16, `$s1` is 17
- `$t0 - $t7` map on hardware registers
8 – 15
 - E.g., `$t0` is 8, `$t1` is 17

Example

`A[300] = h + A[300]`

- Gets compiled to

Example

$A[300] = h + A[300]$

- Gets compiled to

```
lw  $t0,1200($t1) # Temporary reg $t0 gets A[300]
```

```
add $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
```

```
sw  $t0,1200($t1) # Stores h+A[300] back into A[300]
```

Example

$A[300] = h + A[300]$

- Gets compiled to

`lw $t0,1200($t1) # Temporary reg $t0 gets A[300]`

`add $t0,$s2,$t0 # Temporary reg $t0 gets h + A[300]`

`sw $t0,1200($t1) # Stores h+A[300] back into A[300]`

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Instruction encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immediate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

Example

- Decimal

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

- Binary

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

Logical operations

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

Shift

- Shift left by 4

- Before

– 0000 0000 0000 0000 0000 0000 0000 1001 = 9

- After

– 0000 0000 0000 0000 0000 0000 1001 0000 = 144

Encoding shift

- Example

`sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits`

- Shift amount

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Control instructions

Branch instructions

- Branch when equal

 - `beq register1, register2, L1`

 - Go to L1 if register1 equals register2

- Branch when not equal

 - `bne register1, register2, L1`

 - Go to L1 if register1 does *not* equal register2

- Unconditional jump

 - `j L1`

 - Jump to L1

If then .. else ...

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

- Assume that f,g, h, i, and j are in \$s0, \$s1, etc.

If then .. else ...

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

- Assume that f,g, h, i, and j are in \$s0, \$s1, etc.

bne \$s3,\$s4,Else # go to Else if i \neq j

add \$s0,\$s1,\$s2 # f = g + h (skipped if i \neq j)

j Exit # unconditional jump to Exit

Else:

sub \$s0,\$s1,\$s2 # f = g - h (skipped if i = j)

Exit:

Loops

```
while (save[i] == k)
```

```
    i += 1;
```

- Assume that i and k are in \$s3 and \$s5

Loops

```
while (save[i] == k)
```

```
    i += 1;
```

- Assume that i and k are in \$s3 and \$s5

```
Loop: sll $t1,$s3,2
```

```
    add $t1,$t1,$s6    # $t1 = address of save[i]
```

```
    lw $t0,0($t1)      # Temp reg $t0 = save[i]
```

```
    bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
```

```
    addi $s3,$s3,1     # i = i + 1
```

```
    j Loop             # go to Loop
```

```
Exit:
```

Comparisons

- Set on less than

`slt $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4`

- Or with a constraint

`slti $t0,$s2,10 # $t0 = 1 if $s2 < 10`

- Now you can use `slt`, `slti`, `beq`, and `bne` along with `$zero` (register that is always 0)

Branch instructions

- Branch when equal

 - `beq register1, register2, L1`

 - Go to L1 if register1 equals register2

- Branch when not equal

 - `bne register1, register2, L1`

 - Go to L1 if register1 does *not* equal register2

- Unconditional jump

 - `j L1`

 - Jump to L1

Branch instructions

- Branch when equal

 - `beq register1, register2, L1`

 - Go to L1 if register1 equals register2

- Branch when not equal

 - `bne register1, register2, L1`

 - Go to L1 if register1 does *not* equal register2

- Unconditional jump

 - `j L1`

 - Jump to L1

Signed / Unsigned

- The hardware recognizes two formats:

unsigned (corresponding to the C declaration `unsigned int`)

-- all numbers are positive, a 1 in the most significant bit just means it is a really large number

signed (C declaration is `signed int` or just `int`)

-- numbers can be +/- , a 1 in the MSB means the number is negative

This distinction enables us to represent twice as many numbers when we're sure that we don't need negatives

MIPS Instructions

Consider a comparison instruction:

`slt $t0, $t1, $zero`

and \$t1 contains the 32-bit number `1111 01...01`

What gets stored in \$t0?

MIPS Instructions

Consider a comparison instruction:

```
slt $t0, $t1, $zero
```

and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

The result depends on whether \$t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either `slt` or `sltu`

```
slt $t0, $t1, $zero    stores 1 in $t0
```

```
sltu $t0, $t1, $zero   stores 0 in $t0
```

Sign Extension

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand
- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension

So 2_{10} goes from 0000 0000 0000 0010 to
0000 0000 0000 0000 0000 0000 0000 0010

and -2_{10} goes from 1111 1111 1111 1110 to
1111 1111 1111 1111 1111 1111 1111 1110

Registers

- The 32 MIPS registers are partitioned as follows:
 - Register 0 : \$zero always stores the constant 0
 - Regs 2-3 : \$v0, \$v1 return values of a procedure
 - Regs 4-7 : \$a0-\$a3 input arguments to a procedure
 - Regs 8-15 : \$t0-\$t7 temporaries
 - Regs 16-23: \$s0-\$s7 variables
 - Regs 24-25: \$t8-\$t9 more temporaries
 - Reg 28 : \$gp global pointer
 - Reg 29 : \$sp stack pointer
 - Reg 30 : \$fp frame pointer
 - Reg 31 : \$ra return address