Representing Instructions

- Instructions are encoded in binary
  - Called machine code

encoded binary form of the instructions.

- R-type: command defined by func code, opcode = 0
  - func code: 6bit; opcode: 6bit

6bit	5bit	5bit	5bit	5bit	6bit
0	reg1	reg2	reg3	shift amount	opcode

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

8 → 15 $t0 → $t7

16 → 23 $s0 → $s7.

24 → 25 : $t8 - $t9.

Register Name	Common Name	Description
$0	zero	0
$2 - $3	v0 - v1	Result register of functions. Can be temp registers.
$4 - $7	a0 - a3	Arguments
$8 - $15, $24 - $25	t0 - t9	Temporary registers.
$16 - $23, $30	s0 - s8	Saved registers for input/output use. Must have value before use by the call function.
$28	gp	Global pointer
$29	sp	Stack pointer
$31	ra	Return address register, saved by the calling function.

MIPS R-format Instructions

"shift amount"

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

- Instruction fields
  - op: operation code (opcode, 6 bits) (is 0 for R-type)
  - rs: first source register number (5 bits)
  - rt: second source register number (5 bits)
  - rd: destination register number (5 bits)
  - shamt: shift amount (5 bits; 00000 for now)
  - funct: function code (6 bits; extends opcode)

R-format Example

always 0 | always 0

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

func code = 32

```
add $t0, $s1, $s2
```

t0 – t7 : 8 – 15
s0 – s7 : 16 – 23
t8 – t9 = 24 – 25

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

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

Hexadecimal

$aa = 10 \times 16^1 + 10 \times 16^0$
$= 160 + 10 = 170$
2 | 170

- 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

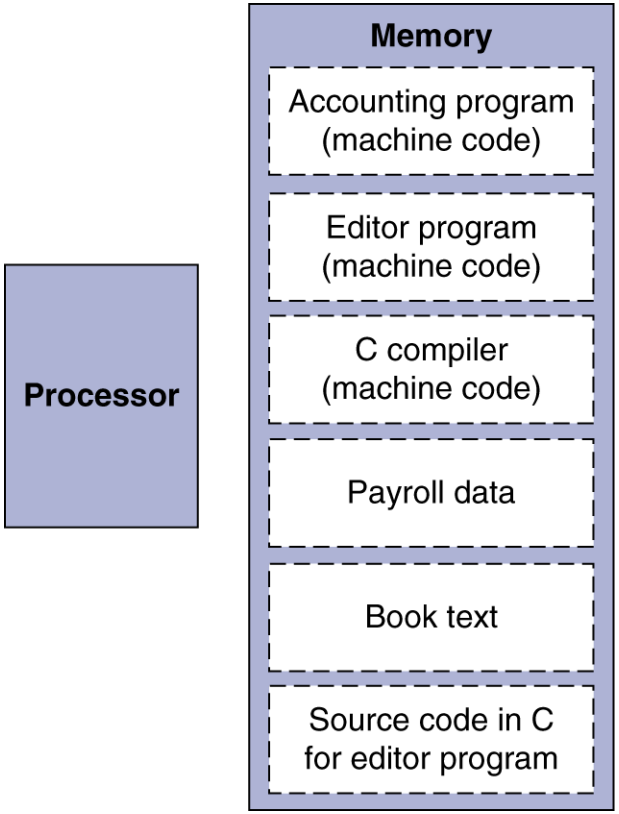
MIPS I-format Instructions

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 $+2^{15} - 1$ 16 bit signed number $(-2^{n-1} \to 2^{n-1}-1)$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Stored Program Computers

The BIG Picture



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

Logical Operations

sll = 2^n.
srl = $/2^n$

- Instructions for bitwise manipulation

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

times number $2^{\wedge n}$
logic devide 2^n

- 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 (times 2^n).
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical (divide 2^n)
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

dest by 1 in selector

```
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

keep these bits by having 1s.

OR Operations

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

dest by 1s in the selector

```
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 ← if NOR with $zero
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

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

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

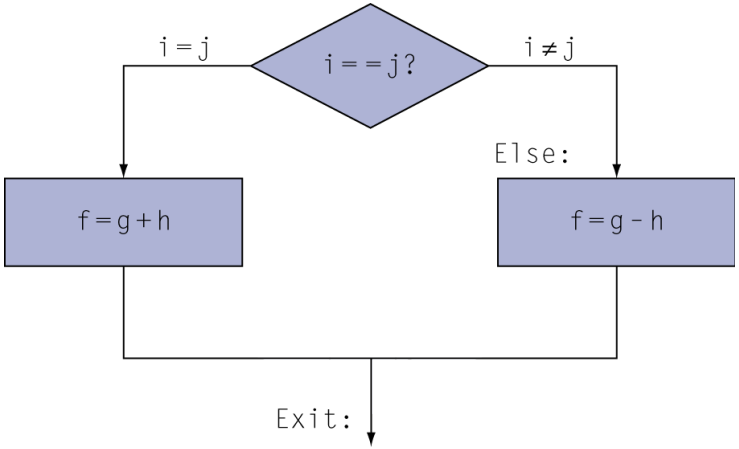
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1` ← label on the right (usually dest is leftmost)
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1` ← label on the right (usually dest is leftmost).
  - if (rs != rt) branch to instruction labeled L1;
- `j L1` ← only 1 parameter
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

```
$s0 $s1 $s2
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

trick use inverted comparison

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

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

  $s6 $s3 $s5 $s3

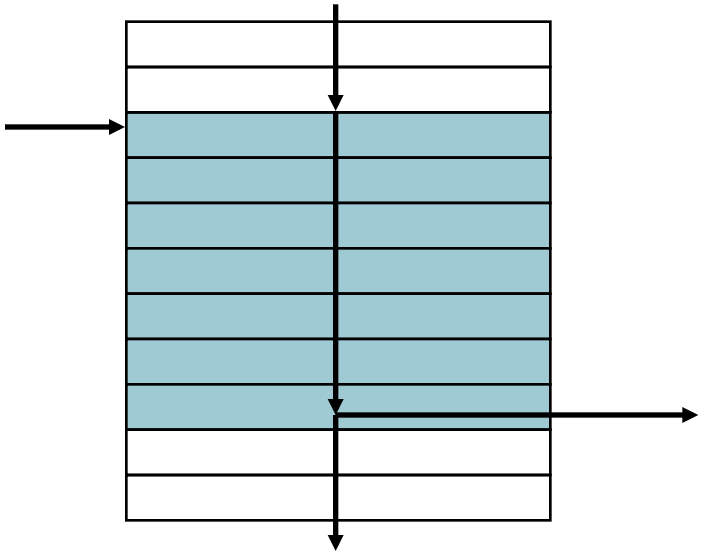
  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2      # $t1 = $s3 × 4
      add  $t1, $t1, $s6    # $t1 = $t1 + $s6
      lw   $t0, 0($t1)      # $t0 = 0($t1)
      bne  $t0, $s5, Exit   # if $t0 <> $s5, exit
      addi $s3, $s3, 1      # $s3 = $s3 + 1
      j    Loop             # jump to "loop"
Exit: …                     # Exit
```

load save[i]

Basic Blocks (not so important)

- 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

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Branch Instruction Design

NO blt, bge

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized! (ie, blt makes all instructions affected to slower, so no “blt”)
- beq and bne are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`

add "u" for unsigned, applied for slt & slti

- 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

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

Register Usage

- $zero : reg 0 ; only contains the value “0”
- $at : reg 1 ; only used by Assembly code
- $v0, $v1: result values (reg’s 2 and 3) result from subroutine
- $a0 – $a3: arguments registers (reg’s 4 – 7) pass args from main prg to subroutine.
- $t0 – $t9: temporaries registers (reg 8 – 15, reg 24 – 25)
  - Can be overwritten by callee
- $s0 – $s7: saved registers (reg 16 -23)
  - Must be saved/restored by callee
- $k0 – $k1: operating system registers (reg 26 -27) kernel.
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel` ← jump into sub routine. (function / procedure)

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

  `jr $ra` ← jump out of sub routine

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

Leaf Procedure Example

- C code:

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

before we modify f in our subroutine

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
```

expand sp 1 word for saving $s0

Save $s0 on stack
before making change to this register

Procedure body

Result

Restore $s0 for caller to use

Return

sometimes people use addui

return $s0 to original value & shrink sp to forget it

jump back to the caller (at jal + 4) addr stored in $ra

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address ← to avoid being overriden
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  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    # branch to L1 if NOT(n<1)
    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
```

recursive call fact(n-1)

Unit 4 Homework

-On P64 fig 2.1 lists all MIPS Instructions and their syntax. On P260 fig 4.12 lists R-type MIPS instructions function codes. add : 32_{10} (100000_2) ; sub : 34_{10} (100010_2) ; and : 36_{10} (100100_2); or : 37_{10} (100101_2) ; slt : 42_{10} (101010_2) ;

-P80 fig 2.5 & P86 fig 2.6:all R-type instruct opcodes: “0000002”; “lw” : “100011_2” (35_{10}); “sw” : “101011_2”(43_{10});

(Q1) op rs rt rd shamt func
0 17 15 8 0 32
00000010 00410010 01000000 00100000

(Q2) srl $t2, $s0, 4. R-type
rd rt sa.

opcode rs rt rd sa func.
0 0 16 10 4. 0x02
000000 00000 10000 01010 00100 000010

+0 .. +9
& 9M 15.

1. Convert MIPS instruction “add $t0, $s1 $s2” to base 2 binary machine codes
2. Convert MIPS instruction ”srl $t2, $s0, 4” to the base 2 binary machine codes
3. What is the MIPS instruction for the following base 10 machine code ?

op (6)	rs (5)	rt (5)	rt(5)	shamt (5)	func(6)
0	8	9	10	0	34

opcode : 0
func : 34 } => This is "sub" R-type MIPS instruction
The formula for sub is
sub $rd, $rs, $rt
=> sub $10, $8, $9
=> sub $t2, $t0, $t1.

4. List all MIPS registers with each register number and its function.
5. Convert following C code to MIPS Assembly, then convert to binary machine codes :

B[250] = ~~B[250]~~ - k ; (B: $t1; k: $s2)

B[250]
lw $t0, 1000($t1)
sub $t0, $t0, $s2
sw $t0, 1000($t1).

lw $t0, 1000($t1)
$rt immediate.($rs)

I-type opcode rs rt immediate.
35 9 8 1000
100011. 01001 01000 0000001111101000

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, register $zero always equals 0, and register $at is reserved by 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 word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
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

FIGURE 2.1 MIPS assembly language revealed in this chapter. This information is also found in Column 1 of the MIPS Reference Data Card at the front of this book.

in the low-order bits of the instruction (see Chapter 2). For branch equal, the ALU must perform a subtraction.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for `beq`, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously.

In Figure 4.12, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the speed of the control unit is often critical to clock cycle time.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the four ALU operation control bits. Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

As a step in designing this logic, it is useful to create a truth table for the interesting combinations of the function code field and the ALUOp bits, as we've

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we "don't care" about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input. See Appendix B.

Design Principle 3: Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

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

The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$. We see that more than 32 registers would be difficult in this format, as the rs and rt fields would each need another bit, making it harder to fit everything in one word.

Let's look at the load word instruction from page 71:

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

Here, 19 (for `$s3`) is placed in the rs field, 8 (for `$t0`) is placed in the rt field, and 32 is placed in the address field. Note that the meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the *destination* register, which receives the result of the load.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first three fields of the R-type and I-type formats are the same size and have the same names; the length of the fourth field in I-type is equal to the sum of the lengths of the last three fields of R-type.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (op) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type). Figure 2.5 shows the numbers used in each field for the MIPS instructions covered so far.

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).

MIPS Encoding Reference

Instruction Encodings

Each MIPS instruction is encoded in exactly one word (32 bits). There are three encoding formats.

Register Encoding (R-Type)

The prototypical R-type instruction is:

```
add $rd, $rs, $rt
```

The semantics of the instruction are:

```
R[d] = R[s] + R[t]
```

This encoding is used for instructions which do not require any immediate data. These instructions receive all their operands in registers. Additionally, certain of the bit shift instructions use this encoding; their operands are two registers and a 5-bit shift amount.

op (6 bits)						rs (5 bits)		rt (5 bits)							
1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
o	o	o	o	o	o	s	s	s	s	s	t	t	t	t	t

rd (5 bits)					shamt (5 bits)			funct (6 bits)							
1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
d	d	d	d	d	a	a	a	a	a	f	f	f	f	f	f

Field	Width	Description
o	6	Instruction opcode. This is 000000 for instructions using this encoding.
s	5	First source register, in the range 0-31.
t	5	Second source register, in the range 0-31.
d	5	Destination register, in the range 0-31.
a	5	Shift amount, for shift instructions.
f	6	Function. Determines which operation is to be performed. Values for this field are documented in the tables at the bottom of this page.

Immediate Encoding (I-Type)

The prototypical I-type instruction looks like:

```
add $rt, $rs, immed
```

The semantics of the addi instruction are;

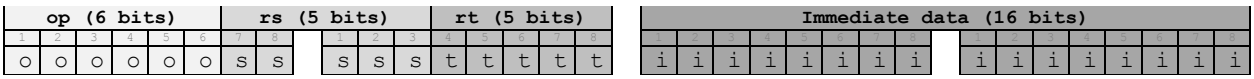
$$R[t] = R[s] + (IR_{15})^{16}\ IR_{15-0}$$

where IR refers to the instruction register, the register where the current instruction is stored. (IR15)16 means that bit B15 of the instruction register (which is the sign bit of the immediate value) is repeated 16 times. This is then followed by IR15-0, which is the 16 bits of the immediate value.

Basically, the semantics says to sign-extend the immediate value to 32 bits, add it (using signed addition) to register R[s], and store the result in register $rt.

This encoding is used for instructions which require a 16-bit immediate operand. These instructions typically receive one operand in a register, another as an immediate value coded into the instruction itself, and place their results in a register. This encoding is also used for load, store, branch, and other instructions so the use of the fields is different in some cases.

Note that the "first" and "second" registers are not always in this order in the assembly language; see "Instruction Syntax" for details.



Field	Width	Description
o	6	Instruction opcode. Determines which operation is to be performed. Values for this field are documented in the tables at the bottom of this page.
s	5	First register, in the range 0-31.
t	5	Second register, in the range 0-31.
i	16	Immediate data. These 16 bits of immediate data are interpreted differently for different instructions. 2's-complement encoding is used to represent a number between -2^{15} and $2^{15}-1$.

Jump Encoding (J-Type)

The prototypical I-type instruction looks like:

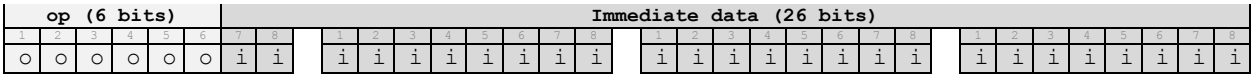
```
j target
```

The semantics of the **j** instruction (**j** means jump) are:

$$PC \leftarrow PC_{31-28}\ IR_{25-0}\ 00$$

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address. The jump instruction will be explained in more detail in a future set of notes.

This encoding is used for jump instructions, which require a 26-bit immediate offset. It is also used for the trap instruction.



Field	Width	Description
o	6	Instruction opcode. Determines which operation is to be performed. Values for this field are documented in the tables at the bottom of this page.

i	26	Immediate data. These 26 bits of immediate data are interpreted differently for different instructions. 2's-complement encoding is used to represent a number between -2^{25} and $2^{25}-1$.

Instruction Syntax

This is a table of all the different types of instruction as they appear in the assembly listing. Note that each syntax is associated with exactly one encoding which is used to encode all instructions which use that syntax.

Encoding	Syntax	Template	Comments
Register	ArithLog	f $d, $s, $t	
	DivMult	f $s, $t	
	Shift	f $d, $t, a	
	ShiftV	f $d, $t, $s	
	JumpR	f $s	
	MoveFrom	f $d	
	MoveTo	f $s	
Immediate	ArithLogI	o $t, $s, i	
	LoadI	o $t, immed32	i is high or low 16 bits of immed32
	Branch	o $s, $t, label	i is calculated as (label - (current + 4)) >> 2
	BranchZ	o $s, label	i is calculated as (label - (current + 4)) >> 2
	LoadStore	o $t, i ($s)	
Jump	Jump	o label	i is calculated as (label - (current + 4)) >> 2
	Trap	o i	

Opcode Table

These tables list all of the available operations in MIPS. For each instruction, the 6-bit opcode or function is shown. The syntax column indicates which syntax is used to write the instruction in assembly text files. Note that which syntax is used for an instruction also determines which encoding is to be used. Finally the operation column describes what the operation does in pseudo-Java plus some special notation as follows:

"`MEM [a]:n`" means the *n* bytes of memory starting with address *a*.
The address must always be aligned; that is, *a* must be divisible by *n*, which must be a power of 2.

"`LB (x)`" means the least significant 8 bits of the 32-bit location *x*.
"`LH (x)`" means the least significant 16 bits of the 32-bit location *x*.
"`HH (x)`" means the most significant 16 bits of the 32-bit location *x*.

"`SE (x)`" means the 32-bit quantity obtained by extending the value *x* on the left with its most significant bit.
"`ZE (x)`" means the 32-bit quantity obtained by extending the value *x* on the left with 0 bits.

Arithmetic and Logical Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
add	100000	32	ArithLog	$d = $s + $t
addu	100001	33	ArithLog	$d = $s + $t
addi	001000	8	ArithLogI	$t = $s + SE(i)
addiu	001001	9	ArithLogI	$t = $s + SE(i)
and	100100	36	ArithLog	$d = $s & $t
andi	001100	12	ArithLogI	$t = $s & ZE(i)
div	011010	26	DivMult	lo = $s / $t; hi = $s % $t
divu	011011	27	DivMult	lo = $s / $t; hi = $s % $t
mult	011000	24	DivMult	hi:lo = $s * $t
multu	011001	25	DivMult	hi:lo = $s * $t
nor	100111	39	ArithLog	$d = ~($s | $t)
or	100101	37	ArithLog	$d = $s | $t
ori	001101	13	ArithLogI	$t = $s | ZE(i)
sll	000000	0	Shift	$d = $t << a
sllv	000100	4	ShiftV	$d = $t << $s
sra	000011	3	Shift	$d = $t >> a
srav	000111	7	ShiftV	$d = $t >> $s
srl	000010	2	Shift	$d = $t >>> a
srlv	000110	6	ShiftV	$d = $t >>> $s
sub	100010	34	ArithLog	$d = $s - $t
subu	100011	35	ArithLog	$d = $s - $t
xor	100110	38	ArithLog	$d = $s ^ $t
xori	001110	14	ArithLogI	$d = $s ^ ZE(i)
Constant-Manipulating Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
lhi	011001	25	LoadI	`HH ($t) = i`

llo	011000	24	LoadI	`LH ($t) = i`
Comparison Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
slt	101010	42	ArithLog	$d = ($s < $t)
sltu	101001	41	ArithLog	$d = ($s < $t)
slti	001010	10	ArithLogI	$t = ($s < SE(i))
sltiu	001001	9	ArithLogI	$t = ($s < SE(i))
Branch Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
beq	000100	4	Branch	if ($s == $t) pc += i << 2
bgtz	000111	7	BranchZ	if ($s > 0) pc += i << 2
blez	000110	6	BranchZ	if ($s <= 0) pc += i << 2
bne	000101	5	Branch	if ($s != $t) pc += i << 2
Jump Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
j	000010	2	Jump	pc += i << 2
jal	000011	3	Jump	$31 = pc; pc += i << 2
jalr	001001	9	JumpR	$31 = pc; pc = $s
jr	001000	8	JumpR	pc = $s
Load Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
lb	100000	32	LoadStore	`$t = SE (MEM [$s + i]:1)`
lbu	100100	36	LoadStore	`$t = ZE (MEM [$s + i]:1)`
lh	100001	33	LoadStore	`$t = SE (MEM [$s + i]:2)`
lhu	100101	37	LoadStore	`$t = ZE (MEM [$s + i]:2)`
lw	100011	35	LoadStore	`$t = MEM [$s + i]:4`
Store Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
sb	101000	40	LoadStore	`MEM [$s + i]:1 = LB ($t)`
sh	101001	41	LoadStore	`MEM [$s + i]:2 = LH ($t)`

sw	101011	43	LoadStore	`MEM [$s + i]:4 = $t`
Data Movement Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
mfhi	010000	16	MoveFrom	$d = hi
mflo	010010	18	MoveFrom	$d = lo
mthi	010001	17	MoveTo	hi = $s
mtlo	010011	19	MoveTo	lo = $s
Exception and Interrupt Instructions				
Instruction	**Opcode/Function**		**Syntax**	**Operation**
trap	011010	26	Trap	Dependent on operating system; different values for immed26 specify different operations. See the list of traps for information on what the different trap codes do.

Opcode Map

ROOT

Table of opcodes for all instructions:

	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	REG		j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	
010								
011	llo	lhi	trap					
100	lb	lh		lw	lbu	lhu		
101	sb	sh		sw				
110								
111								

REG

Table of function codes for register-format instructions:

	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010	mfhi	mthi	mflo	mtlo				

011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

Register Name	Common Name	Description
$0	zero	Always has the value 0. Any writes to this register are ignored.
$1	at	Assembler temporary.
$2-$3	v0-v1	Function result registers. Functions return integer results in v0, and 64-bit integer results in v0 and v1 when using 32-bit registers. In cases where floating-point hardware is not present, or when compiler options enable floating-point emulation, functions return single precision floating-point results in v0 and double precision floating-point results in v0 and v1 when using 32-bit registers. v0 and v1 can be temporary registers. Not preserved across function calls.
$4-$7	a0-a3	Function argument registers that hold the first four words of integer type arguments. Functions use these registers to hold floating-point arguments. When floating-point hardware is not present, or compiler options enable floating-point emulation, functions use a0 to hold the first single precision floating-point argument and a1 to hold the second single precision floating-point argument. Functions use a0-a1 for the first double precision floating-point argument, and a2-a3 to hold the second double precision floating-point argument. Not preserved across function calls.
$8-$15, $24-$25	t0-t9	Temporary registers you can use as you want. Not preserved across function calls.
$16-$23, $30	s0-s8	Saved registers to use freely. Preserved across function calls. These registers must be saved before use by the called function.
$26-$27	k0-k1	Reserved for use by the operating system kernel and for exception return.

$28	gp	Global pointer. Not used in Windows CE and may be used as save register for called functions.
$29	sp	Stack pointer.
$31	ra	Return address register, saved by the calling function. Available for use after saving.
$f0	n/a	Function return register used to return float and double values from function calls.
($f12, $f13) and ($f14, $f15)	n/a	Two pairs of registers used to pass float and double valued parameters to functions. Pairs of registers are parenthesized because they have to pass double values. To pass float values, only $f12 and $f14 are used.

The following list contains additional information on floating-point registers:

- In MIPS ISAs I, II, and in MIPS III and up ISAs running in 32-bit mode, only $f4, $f6, $f8, $f10, $f16, and $f18 temporary registers are available.
  When manipulating these registers with double precision instructions, the high-order 32-bits are in the implied odd register. The odd registers are not directly accessible.
- Permanent registers $f20, $f22, $f24, $f26, $f28, and $f30 are registers where values are preserved across function calls.
- In MIPS architectures III and up running in 64-bit mode, the following registers are also available as temporary registers: $f1, $f3, $f5, $f7, $f9, $f11, $f17, $f19, $f21, $f23, $f25, $f27, $f29, $f31.