ISA Part III
Logical and Shift Operations
Instruction Representation

MIPS Instructions (Quick Summary)

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

Overview

- **Logical Instructions**
- **Shifts**
- **Instruction Formats**

Bitwise Operations

- **Up until now, we've done arithmetic (`add`, `sub`,`addi`), memory access (`lw` and `sw`), and branches and jumps.**
- **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 (or groups of bits) rather than the whole.**
- **Introduce two new classes of instructions:**
  - Logical Operators
  - Shift Instructions

Logical Operators

- **Two basic logical operators:**
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1

Logical Operators

- **Two basic logical operators:**
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1

- **Truth Table: standard table listing all possible combinations of inputs and resultant output for each**

- **Truth Table for AND and OR**

A	B	AND	OR
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**1**

Logical Operators

- **Instruction Names:**
  - `and`, `or`: Both of these expect the third argument to be a register
  - `andi`, `ori`: Both of these expect the third argument to be 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.**

Uses for Logical Operators

- **Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.**

- **This can be used to create a mask.**

  – Example:

  1011 0110 1010 0100 0011 1101 1001 1010

  0000 0000 0000 0000 0000 1111 1111 1111

  – The result of anding these two is:

  0000 0000 0000 0000 0000 1101 1001 1010

- **The second bit string in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bit string by masking out the rest of the string (e.g. setting it to all 0s).**

Uses for Logical Operators

- **Thus, the `and` operator can be used to set certain portions of a bit string to 0s, while leaving the rest alone.**
  - In particular, if the first bit string in the above example were in $t0, then the following instruction would mask it:

```
andi    $t0,$t0,0xFFF
```

- **Similarly, note that `or`ing a bit with 1 produces a 1 at the output while `or`ing a bit with 0 produces the original bit.**

- **This can be used to force certain bits of a string to 1s.**
  - For example, if $t0 contains 0x12345678, then after this instruction:

```
ori  $t0, $t0, 0xFFFF
```

  - $t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

Shift Instructions (1/3)

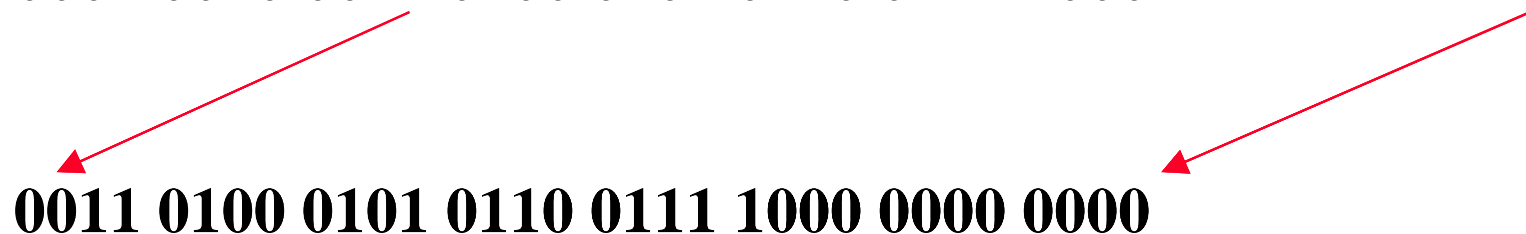
- **Move (shift) all the bits in a word to the left or right by a number of bits.**
  - Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- **Example: shift left by 8 bits**

0001 0010 0011 0100 0101 0110 0111 1000



0011 0100 0101 0110 0111 1000 0000 0000

Shift Instructions (2/3)

- **Shift instruction syntax:**

  1 2,3,4

  – Where

  1) operation name

  2) register that will receive value

  3) first operand (register)

  4) shift amount (constant <= 32)

MIPS shift instructions:

1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (3/3)

- **Example: shift right arithmetic by 8 bits**

```
 0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110
```

- **Example: shift right arithmetic by 8 bits**

```
1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110
```

- **Example: shift right logical by 8 bits**

```
1001 0010 0011 0100 0101 0110 0111 1000

0000 0000 1001 0010 0011 0100 0101 0110
```

Uses for Shift Instructions (1/4)

- **Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0. Simply use:**

```
andi    $t0,$t0,0xff.
```

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

```
andi    $t0,$t0,0xff00.
```

But then we still need to shift to the right by 8 bits...

Uses for Shift Instructions (2/4)

- **Could use instead:**

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 (3/4)

- **In binary:**

  Multiplying by 2 is same as shifting left by 1:

  $11_2 \times 10_2 = 110_2$

  $1010_2 \times 10_2 = 10100_2$

  Multiplying by 4 is same as shifting left by 2:

  $11_2 \times 100_2 = 1100_2$

  $1010_2 \times 100_2 = 101000_2$

  Multiplying by 2^n is same as shifting left by n

- **In decimal:**

  Multiplying by 10 is same as shifting left by 1:

  $714_{10} \times 10_{10} = 7140_{10}$

  $56_{10} \times 10_{10} = 560_{10}$

  Multiplying by 100 is same as shifting left by 2:

  $714_{10} \times 100_{10} = 71400_{10}$

  $56_{10} \times 100_{10} = 5600_{10}$

  Multiplying by 10^n is same as shifting left by n

Uses for Shift Instructions (4/4)

- **Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:**

  `a *= 8;` (in C)

  would compile to:

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

- **Likewise, shift right to divide by powers of 2**
  - remember to use `sra (shift arithmetic)`

Big Idea: Stored-program Concept

- **Computers built on 2 key principles:**

  1) instructions are represented as numbers.

  2) therefore, entire programs can be stored in memory to be read or written just like numbers (data).

- **Simplifies SW/HW of computer systems:**

  1. Memory technology for data also used for programs.
  2. Data and Instructions are just 1's and 0's.

Result #1: Everything Addressed

- **Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words**
  - both branches and jumps use these
- **C pointers are just memory addresses: they can point to anything in memory**
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- **One register keeps address of instruction being executed: ‘Program Counter’ (PC)**
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, which is better

Result #2: Binary Compatibility

- **Programs are distributed in binary form**
  - Programs bound to specific instruction set
  - Different version for Macintosh and IBM PC
- **New machines want to run old programs (‘binaries’) as well as programs compiled to new instructions**
- **Leads to instruction set evolving over time**
- **Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); Could still run program from 1981 PC today**

Instructions As Numbers

- **Currently all data we work with is in 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 meaningless.
  - MIPS wants simplicity: since data is in words, make instructions be words...
- **One word is 32 bits, so 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**
  - **I-format**
  - **J-format**

Format Instructions

- **J-format: used for `j` and `jal`**
- **I-format: used for instructions with immediates, `lw` and `sw` (since the offset counts as an immediate), and the branches (`beq` and `bne`),**

  (But not the shift instructions; Later)

- **R-format: used for all other instructions**
- **It will soon become clear why the instructions have been partitioned in this way.**

R-Format Instructions (1/3)

- **Define 'fields' of the following number of bits each:**

6	5	5	5	5	6

- **For simplicity, each field has a name:**

opcode	rs	rt	rd	shamt	funct

- **Important: Each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.**

Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.

R-Format Instructions (2/3)

- **What do these field integer values tell us?**
  - `opcode`: partially specifies what instruction it is (Note: This number is equal to 0 for all R-Format instructions.)
  - `funct`: combined with `opcode`, this number exactly specifies the instruction

- **More fields:**
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand (note that name is misleading)
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

R-Format Instructions (3/3)

- **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.
  - The word 'generally' was used because there are exceptions, such as:
    - `mult` and `div` have nothing important in the `rd` field since the dest registers are `hi` and `lo`
    - `mfhi` and `mflo` have nothing important in the `rs` and `rt` fields since the source is determined by the instruction
- **Final field:**
  - `shamt`: This field 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 (so it can represent the numbers 0-31).
  - This field is set to 0 in all but the shift instructions.
- **For a detailed description of field usage for each instruction, see back cover of textbook.**

R-Format Example (1/2)

- **MIPS Instruction:**

```
add   $8,$9,$10
```

`opcode` = 0 (look up in table)

`funct` = 32 (look up in table)

`rs` = 9 (first *operand*)

`rt` = 10 (second *operand*)

`rd` = 8 (destination)

`shamt` = 0 (not a shift)

R-Format Example (2/2)

- **MIPS Instruction:**

```
add   $8,$9,$10
```

decimal representation:

0	9	10	8	0	32

binary representation:

000000	01001	01010	01000	00000	100000

Called a Machine Language Instruction

Reading The Table

- **MIPS Instruction:**

```
add   $8,$9,$10
```

10	16	op(31:26)	10	funct(5:0)	10	funct(5:0)
0	00		0	sll	0	add.f
1	01		1		1	sub.f
2	02	j	2	srl	2	mul.f
3	03	jal	3	sra	3	div.f
4	04	beq	4	sllv	4	
5	05	bne	5		5	abs.f
6	06	blez	6	srlv	6	mov.f
7	07	bgtz	7	srav	7	neg.f
8	08	addi	8	jr	8	
9	09	addiu	9	jalr	9	
10	0a	slti	10		10	
11	0b	sltiu	11		11	
12	0c	andi	12	syscall	12	
13	0d	ori	13	break	13	
14	0e	xori	14		14	
15	0f	lui	15		15	
16	10	z = 0	16	mfhi	16	
17	11	z = 1	17	mthi	17	
18	12	z = 2	18	mflo	18	
19	13	z = 3	19	mtlo	19	
20	14		20		20	
21	15		21		21	
22	16		22		22	
23	17		23		23	
24	18		24	mult	24	
25	19		25	multu	25	
26	1a		26	div	26	
27	1b		27	divu	27	
28	1c		28		28	
29	1d		29		29	
30	1e		30		30	
31	1f		31		31	
32	20	lb	32	add	32	cvt.s.f
33	21	lh	33	addu	33	cvt.d.f
34	22	lwl	34	sub	34	
35	23	lw	35	subu	35	
36	24	lbu	36	and	36	cvt.w.f
37	25	lhu	37	or	37	
38	26	lwr	38	xor	38	
39	27		39	nor	39	
40	28	sb	40		40	
41	29	sh	41		41	
42	2a	swl	42	slt	42	
43	2b	sw	43	sltu	43	
44	2c		44		44	
45	2d		45		45	
46	2e	swr	46		46	
47	2f		47		47	
48	30	lwc0	48		48	c.f.f
49	31	lwc1	49		49	c.un.f
50	32	lwc2	50		50	c.eq.f
51	33	lwc3	51		51	c.ueq.f
52	34		52		52	c.olt.f
53	35		53		53	c.ult.f
54	36		54		54	c.ole.f
55	37		55		55	c.ule.f
56	38	swc0	56		56	c.st.f
57	39	swc1	57		57	c.ngle.f
58	3a	swc2	58		58	c.seq.f
59	3b	swc3	59		59	c.ngl.f
60	3c		60		60	c.lt.f
61	3d		61		61	c.nge.f
62	3e		62		62	c.le.f
63	3f		63		63	c.ngt.f

	rs (25:21)		funct (4:0)		rt (20:16)
0	mfcz	0		0	bltz
1		1	tlbr	1	bgez
2	cfcz	2	tlbwi	2	
3		3		3	
4	mtcz	4		4	
5		5		5	
6	ctcz	6	tlbwr	6	
7		7		7	
8		8	tlbp	8	
9		9		9	
10		10		10	
11		11		11	
12		12		12	
13		13		13	
14		14		14	
15		15		15	
16	copz	16	rfe	16	bltzal
17	copz	17		17	bgezal
18		18		18	
19		19		19	
20		20		20	
21		21		21	
22		22		22	
23		23		23	
24		24		24	
25		25		25	
26		26		26	
27		27		27	
28		28		28	
29		29		29	
30		30		30	
31		31		31	

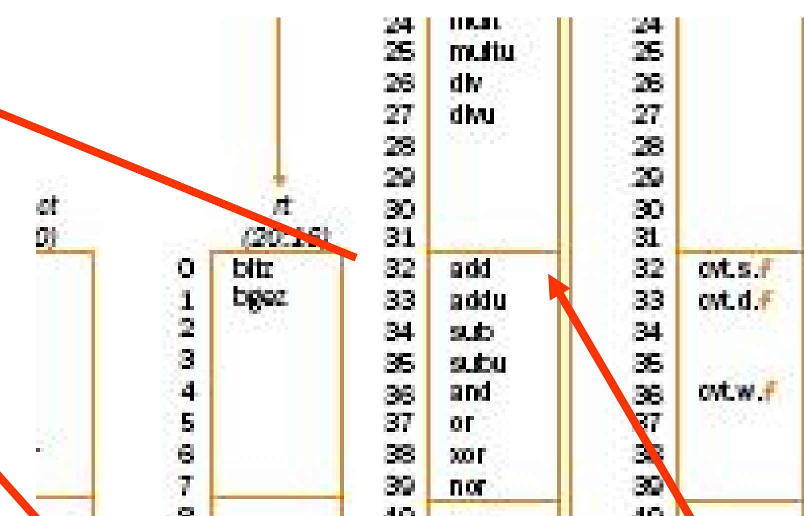
	(16:16)
0	bczf
1	bczt

if z = 0

if z = 1, f = d

if z = 1, f = s

FIGURE A.19 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10 and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for 6 op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses "*f*" to mean "s" if rs = 16 and op = 17 or "d" if rs = 17 and op = 17. The second field (rs) uses "*z*" to mean "0", "1", "2", or "3" if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if *z* = 0, the operations are specified in the fourth field (bits 4 to 0); if *z* = 1, then the operations are in the last field with *f* = s. If rs = 17 and *z* = 1, then the operations are in the last field with *f* = d. *(page A-54)*



0	9	10	8	0	32

I-format Instructions

- **What about instructions with immediates?**
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this.
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise.
- **Define new instruction format that is partially consistent with r-format:**
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-format Instructions

- **Define 'fields' of the following number of bits each:**

6	5	5	16

- **Again, each field has a name:**

opcode	rs	rt	immediate

- **Key Concept: Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location.**

I-format Instructions

- **The immediate field:**
  - `addi`, `slti`, `slitu`, the immediate is sign-extended to 32 bits. Thus, it's treated as a signed integer.
  - 16 bits - can be used to represent immediate up to 2^{16} different values.
  - This is large enough to handle the offset in a typical `lw` or `sw`, plus a vast majority of values that will be used in the `slti` instruction.

I-format Example

- **MIPS Instruction:**

`addi $21,$22,-50`

`opcode` = 8 (look up in table)

`rs` = 22 (register containing operand)

`rt` = 21 (target register)

`immediate` = -50 (by default, this is decimal)

decimal representation:

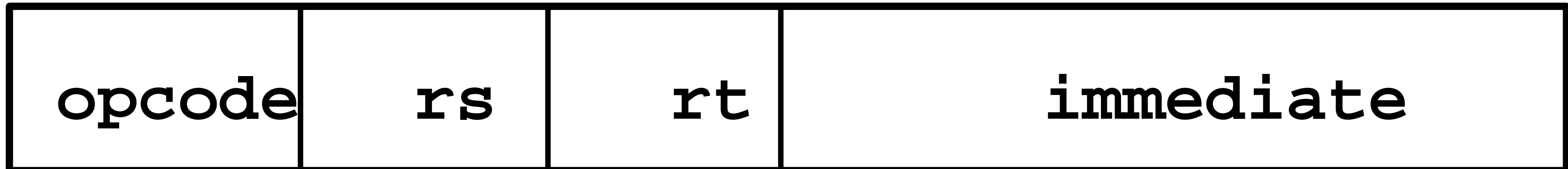
8	22	21	-50

binary representation:

001000	10110	10101	1111111111001110

Branches: PC-relative Addressing

- **Use I-Format**



- **`opcode` specifies `beq` v. `bne`**
- **`Rs` and `Rt` specify registers to compare**
- **What can `immediate` specify?**
  - **`Immediate` is only 16 bits**
  - **PC is 32-bit pointer to memory**
  - **So `immediate` cannot specify entire address to branch to.**

Branches: PC-relative Addressing

- **How do we usually use branches?**
  - Answer: `if-else`, `while`, `for`
  - Loops are generally small: typically up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches.
- **Conclusion: though we may want to branch to anywhere in memory, a single branch will generally change the PC by a very small amount.**

Branches: PC-relative Addressing

- **Solution: pc-relative addressing**

- **Let the 16-bit `immediate` field be a signed two's complement integer to be *added* to the PC if we take the branch.**

- **Now we can branch +/- 2^{15} bytes from the PC, which should be enough to cover any loop.**

- **Any ideas to further optimize this?**

Branches: PC-relative Addressing

- **Note: instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary).**
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the `immediate` in words.
- **Now, we can branch +/- 2^{15} words from the PC (or +/- 2^{17} bytes), so we can handle loops 4 times as large.**

Branches: PC-relative Addressing

- **Final calculation:**
  - If we don't take the branch:

  PC = PC + 4.

  - If we do take the branch:

  PC = (PC + 4) + (imm`ediate * `4).

  - Observations.
    - `Immediate` field specifies the number of words to jump, which is simply the number of instructions to jump.
    - `Immediate` field can be positive or negative.
    - Due to hardware, add `immediate` to (PC+4), not to PC; Will be clearer why later in course.

Branch Example (1/3)

- **MIPS Code:**

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

- **Branch is I-Format:**

`opcode` = 4 (look up in table)

`rs` = 9 (first operand)

`rt` = 0 (second operand)

`immediate` = ???

Branch Example (2/3)

- **MIPS Code:**

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

- **`Immediate` Field:**
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch.
  - In this case, `immediate` = 3

Branch Example (3/3)

- **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

Things to Remember

- Simplifying MIPS: define instructions to be same size as data (one word) so that they can use the same memory (can use `lw` and `sw`).

  - Machine language instruction: 32 bits representing a single instruction.

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		

Computer actually stores programs as a series of these.

I-Format Problems (1/3)

- **Problem 1:**
  - Chances are that `addi`, `lw`, `sw` and `slti` will use immediates small enough to fit in the immediate field.
  - What if too big?
    - We need a way to deal with a 32-bit immediate in *any* I-format instruction.

I-Format Problems (2/3)

- **Possible Solutions to Problem 1:**
  - Handle it in software
  - Don't change the current instructions: instead, add a new instruction to help out

- **New instruction:**

  lui `register, immediate`

  - stands for Load Upper Immediate
  - takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
  - sets lower half to 0s

I-Format Problems (3/3)

- **Solution to Problem 1 (continued):**
  - So how does `lui` help us?
  - Example:

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

becomes:

```
lui     $at, 0xABAB
ori     $at, $at, 0xCDCD
add     $t0,$t0,$at
```

  - Now each I-format instruction has only a 16-bit immediate.