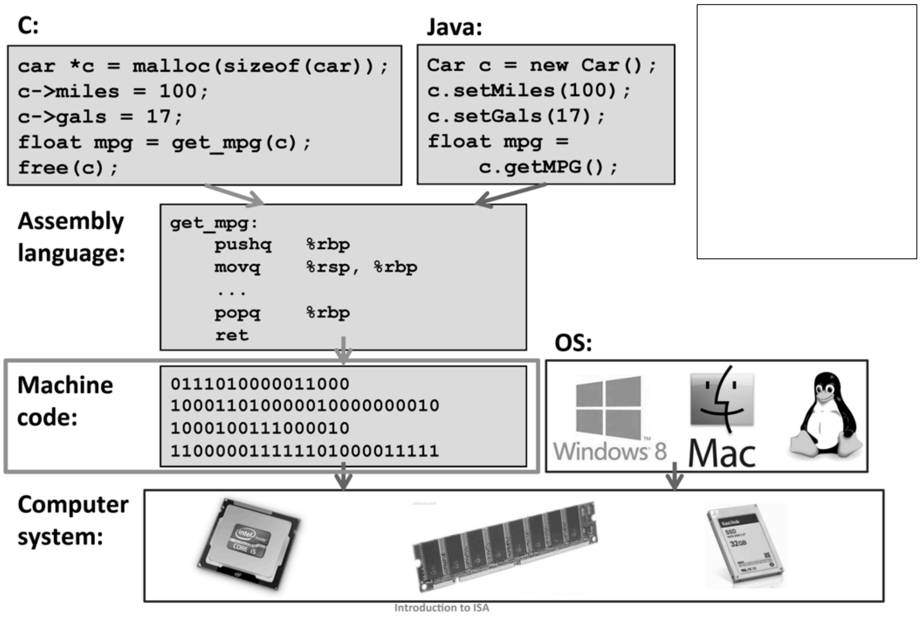
CS 305
MIPS ISA

Prof Umesh Bellur

Instruction Set Architecture



C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
      c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

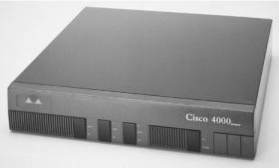
OS:

Computer system:

Introduction to ISA

MIPS

- In this class, we’ ll use the MIPS instruction set architecture (ISA) to illustrate concepts in assembly language and machine organization
  - Of course, the concepts are not MIPS-specific
  - MIPS is just convenient because it is real, yet simple (unlike x86)
- The MIPS ISA is still used in many places today. Primarily in embedded systems, like:
  - Various routers from Cisco
  - Game machines like the Nintendo 64 and Sony Playstation 2



MIPS Design Principles

- **Simplicity favors regularity**
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- **Good design demands good compromises**
  - three instruction formats
- **Smaller is faster**
  - limited instruction set
  - compromise on number of registers in register file
  - limited number of addressing modes
- **Make the common case fast**
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

MIPS

- All instructions have
  - <= 1 arithmetic op
  - <= 1 memory access
  - <= 2 register reads
  - <= 1 register write
  - <= 1 branch
  - It needs a small, fixed amount of hardware.
- Instructions operate on memory or registers not both
  - "Load/Store Architecture"
- Decoding is easy
  - Uniform opcode location
  - Uniform register location
  - Always 4 bytes -> the location of the next PC is to know.
- Uniform execution algorithm
  - Fetch
  - Decode
  - Execute
- Compiling is easy
  - No complex instructions to reason about
  - No special registers
- The HW is simple
- 33 instructions(MIPS I) can run complex programs.

Vs x86

- Many, many instruction formats. Variable length (4 to 15 bytes).
- Many complex rules about which register can be used when, and which addressing modes are valid where.
- Very complex instructions
  - Combined memory/arithmetic.
  - Special-purpose registers.
- 100s of instructions.
  - Implementing x86 correctly is almost intractable

Vs ARM

- ARM is somewhere in between
  - Four instruction formats. Fixed length.
  - General purpose registers (except the condition codes)
  - Moderately complex instructions, but they are still "regular" -- all instructions look more or less the same.
- ARM targeted embedded systems
  - Code density is important
  - Performance (and clock speed) is less critical
  - Both of these argue for more complex instructions.
  - But they can still be regular, easy to decode, and crafted to minimize hardware complexity
- Implementing an ARM processor is also tractable, but it would be harder than MIPS

What you will need to learn

- You must become "fluent" in MIPS assembly:
  - Translate from C to MIPS and MIPS to C
- Example problem from a previous mid-term:

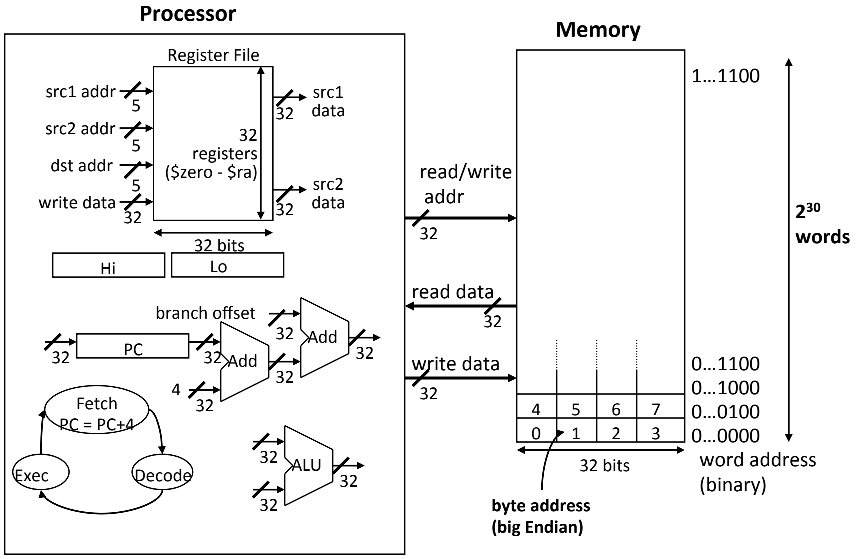
Question 3: Write a recursive function (30 points)

Here is a function pow that takes two arguments (n and m, both 32-bit numbers) and returns n^m (i.e., n raised to the m^{th} power).

```
int
pow(int n, int m) {
   if (m == 1)
      return n;
   return n * pow(n, m-1);
}
```

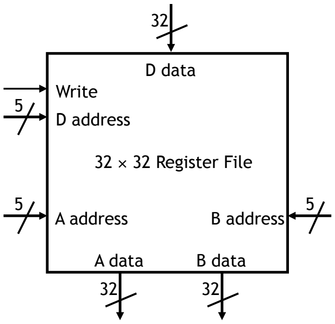
Translate this into a MIPS assembly language function.

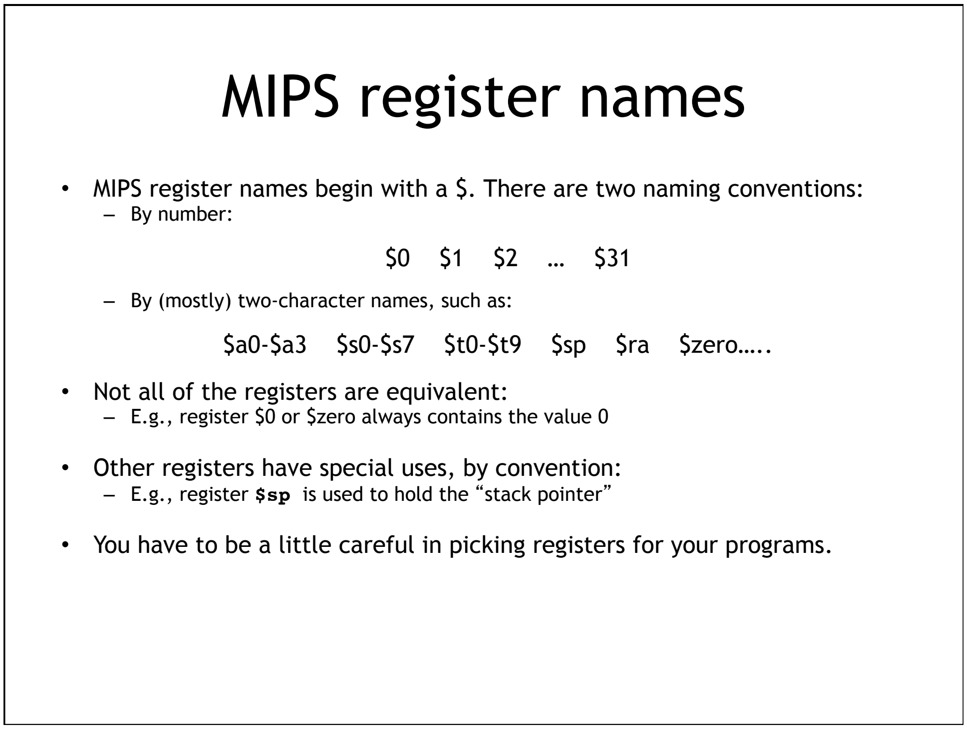
MIPS Organization



MIPS registers

- MIPS processors have **32 general purpose registers**, each of which holds **a 32-bit value**.
  - Register addresses are 5 bits long.
  - The data inputs and outputs are 32-bits wide.
  - There are 2 special registers called Hi and Lo for multiplication
- More registers might seem better, but there is a limit to the goodness.
  - It' s more expensive, because of both the registers themselves as well as the decoders and muxes needed to select individual registers.
  - Instruction lengths may be affected, as we' ll see in the future.





MIPS register names

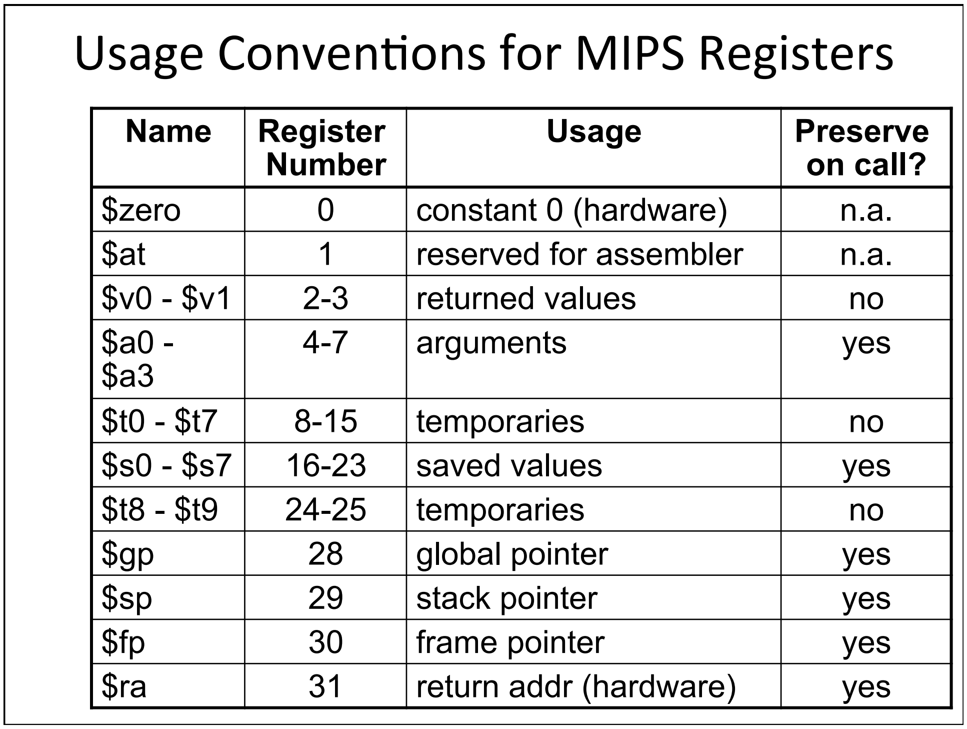
- MIPS register names begin with a $. There are two naming conventions:
  - By number:

    $0 $1 $2   … $31

  - By (mostly) two-character names, such as:

    $a0-$a3 $s0-$s7 $t0-$t9 $sp $ra $zero…..

- Not all of the registers are equivalent:
  - E.g., register $0 or $zero always contains the value 0
- Other registers have special uses, by convention:
  - E.g., register **`$sp`** is used to hold the “stack pointer”
- You have to be a little careful in picking registers for your programs.



Usage Conventions for MIPS Registers

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS Instruction Formats

- R-type, 3 register operands

opcode	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

- I-type, 2 register operands and 16-bit immediate operand

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- J-type, 26-bit immediate operand

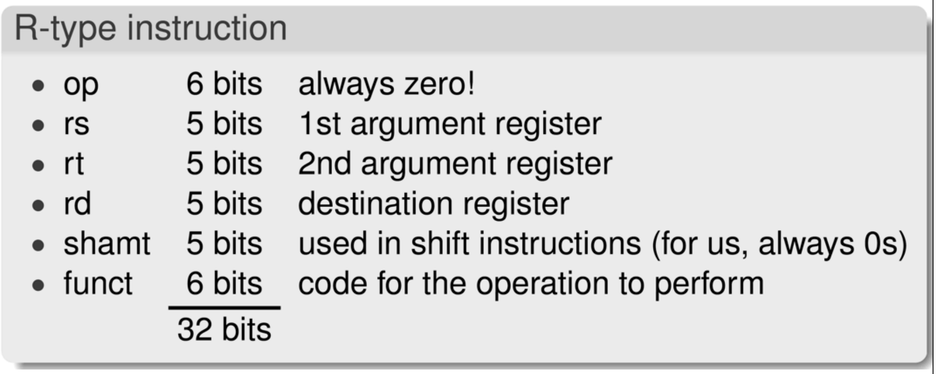
opcode	immediate
6-bit	26-bit

J-type

- Simple Decoding
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b'00)
  - format and fields easy to extract in hardware

R type instructions

R:	000000	rs	rt	rd	shamt	funct



R-type instruction

• op	6 bits	always zero!
• rs	5 bits	1st argument register
• rt	5 bits	2nd argument register
• rd	5 bits	destination register
• shamt	5 bits	used in shift instructions (for us, always 0s)
• funct	6 bits	code for the operation to perform
	32 bits	

Assembling an R type instruction

`add $t1, $t2, $t3`

rs	=	10	(`$t2` = `$10`)
rt	=	11	(`$t3` = `$11`)
rd	=	9	(`$t1` = `$9`)
funct	=	32	(look up function code for **`add`**)
shamt	=	0	(not a shift instruction)

000000	10	11	9	0	32
000000	01010	01011	01001	00000	100000

0000 0001 0100 1011 0100 1000 0010 0000

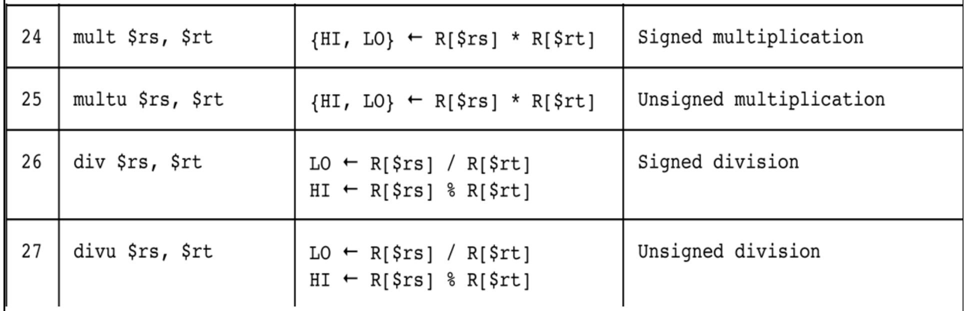
0x014B4820

R type instructions – Arithmetic and Logical

32	add $rd, $rs, $rt	R[$rd] ← R[$rs] + R[$rt]	Exception on signed overflow
33	addu $rd, $rs, $rt	R[$rd] ← R[$rs] + R[$rt]	
34	sub $rd, $rs, $rt	R[$rd] ← R[$rs] – R[$rt]	Exception on signed overflow
35	subu $rd, $rs, $rt	R[$rd] ← R[$rs] – R[$rt]	
36	and $rd, $rs, $rt	R[$rd] ← R[$rs] & R[$rt]	
37	or $rd, $rs, $rt	R[$rd] ← R[$rs] | R[$rt]	
38	xor $rd, $rs, $rt	R[$rd] ← R[$rs] ^ R[$rt]	

These all require **three** register operands with **Rd** always indicating the destination register

More R type Arithmetic instructions



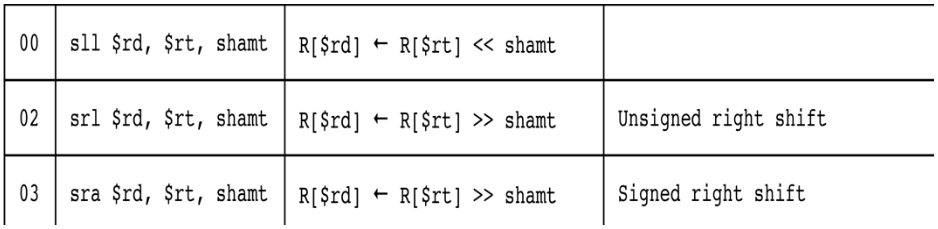
24	mult $rs, $rt	{HI, LO} ← R[$rs] * R[$rt]	Signed multiplication
25	multu $rs, $rt	{HI, LO} ← R[$rs] * R[$rt]	Unsigned multiplication

26	div $rs, $rt	LO ← R[$rs] / R[$rt]
		HI ← R[$rs] % R[$rt]
			Signed division

27	divu $rs, $rt	LO ← R[$rs] / R[$rt]
		HI ← R[$rs] % R[$rt]
			Unsigned division

Note: These are 2 register operations only.
The special purpose Hi and Lo registers are used for the result.

R type Shift operations using SHAMT (2 register operations)

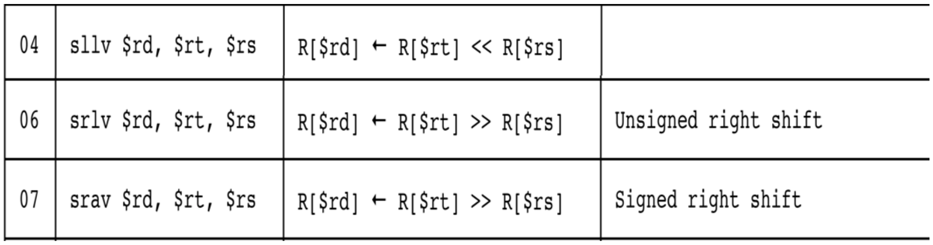


00	sll $rd, $rt, shamt	R[$rd] ← R[$rt] << shamt	
02	srl $rd, $rt, shamt	R[$rd] ← R[$rt] >> shamt	Unsigned right shift
03	sra $rd, $rt, shamt	R[$rd] ← R[$rt] >> shamt	Signed right shift

Why does Left Shift not have signed and unsigned versions??

They mean the same thing since zeroes are shifted in from the right.

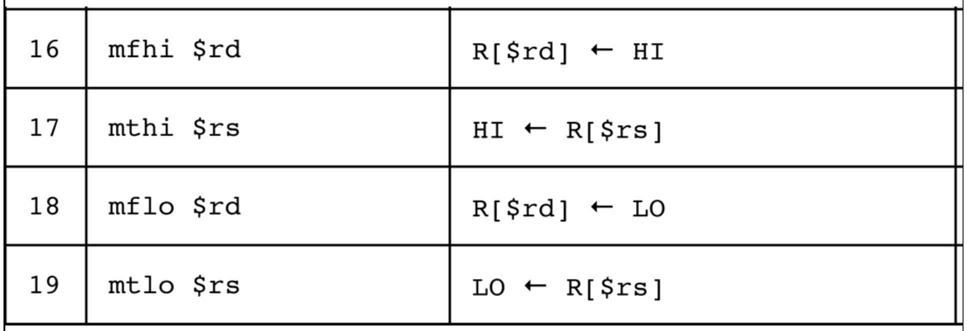
R type Shift ops for variable number of shift bits (3 register)



04	sllv $rd, $rt, $rs	R[$rd] ← R[$rt] << R[$rs]	
06	srlv $rd, $rt, $rs	R[$rd] ← R[$rt] >> R[$rs]	Unsigned right shift
07	srav $rd, $rt, $rs	R[$rd] ← R[$rt] >> R[$rs]	Signed right shift

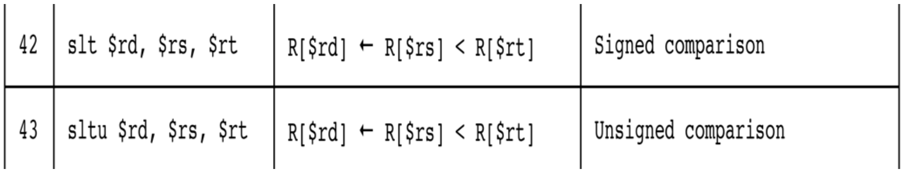
Shift Left Logical Variable (SLLV): The contents of 32-bit word of Rt are shifted left, inserting zeroes into the emptied bits; the result word is placed in Rd. The bit shift count is specified by the **low-order five bits of GPR rs**

R Type Move ops (1 register)



16	mfhi $rd	R[$rd] ← HI
17	mthi $rs	HI ← R[$rs]
18	mflo $rd	R[$rd] ← LO
19	mtlo $rs	LO ← R[$rs]

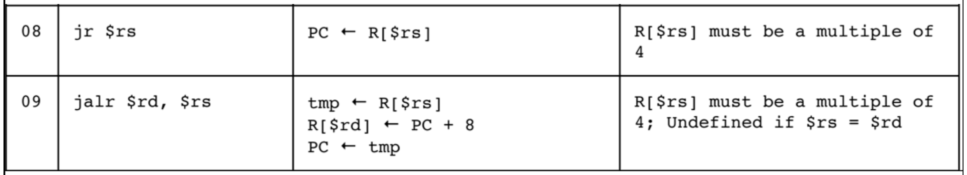
R Type Comparison



42	slt $rd, $rs, $rt	R[$rd] ← R[$rs] < R[$rt]	Signed comparison
43	sltu $rd, $rs, $rt	R[$rd] ← R[$rs] < R[$rt]	Unsigned comparison

<u>SLT:</u> Set if Less Than

R type Jump Instructions



08	jr $rs	PC ← R[$rs]	R[$rs] must be a multiple of 4

09	jalr $rd, $rs	tmp ← R[$rs]
		R[$rd] ← PC + 8
		PC ← tmp
			R[$rs] must be a multiple of 4; Undefined if $rs = $rd

All branches have an **architectural delay of one instruction**. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place.

This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31 The default register for GPR rd , if omitted in the assembly language instruction, is GPR 31.

Larger expressions

- More complex arithmetic expressions will require multiple operations at the instruction set level.

t0 = (t1 + t2) × (t3 - t4)

```
add $t0, $t1, $t2  # $t0 contains $t1 + $t2
sub $s0, $t3, $t4  # Temporary value $s0 = $t3 - $t4
mul $t0, $t0, $s0  # $t0 contains the final product
```

- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination.
  - In this example, we could re-use $t3 instead of introducing $s0.
  - But be careful not to modify registers that are needed again later.

Immediate operands

- The instructions we' ve seen so far expect register operands. How do you get data into registers in the first place?
  - Some MIPS instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register. For example, here is the immediate add instruction, addi:

```
addi  $t0, $t1, 4    # $t0 = $t1 + 4
```

  - Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi  $t0, $0, 4     # $t0 = 4
```

op	rs	rt	16 bit immediate

I format

- MIPS is still considered a load/store architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.

❑ The constant is kept inside the instruction itself!

- **Immediate format limits values to the range $+2^{15}–1$ to -2^{15}**

I Type Instructions

I:	op	rs	rt	address/immediate

I-type instruction

• op	6 bits	code for the operation to perform
• rs	5 bits	1st argument register
• rt	5 bits	destination or 2nd argument register
• imm	16 bits	constant value embedded in instruction
	32 bits	

Note the destination register is second in the machine code!

Assembling an I Type Instruction

```
addi $t4, $t5, 67
```

op	rs	rt	address/immediate

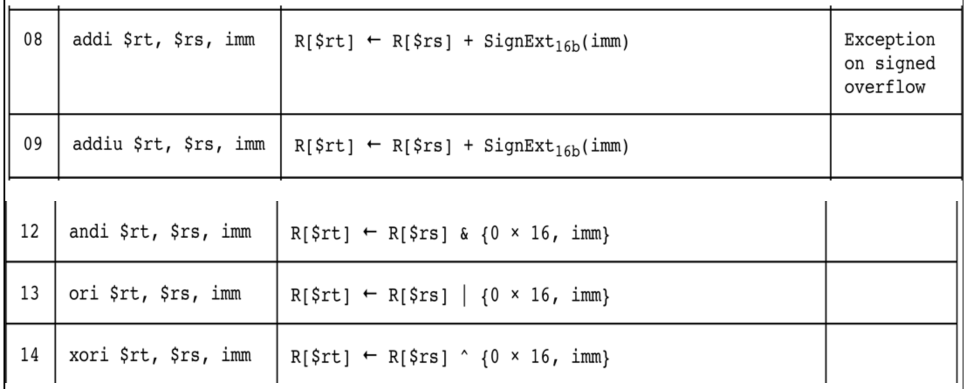
op	=	8	(look up op code for `addi`)
rs	=	13	(`$t5` = `$13`)
rt	=	12	(`$t4` = `$12`)
imm	=	67	(constant value)

8	13	12	67
`001000`	`01101`	`01100`	`0000 0000 0100 0011`

```
0010 0001 1010 1100 0000 0000 0100 0011
```

```
0x21AC0043
```

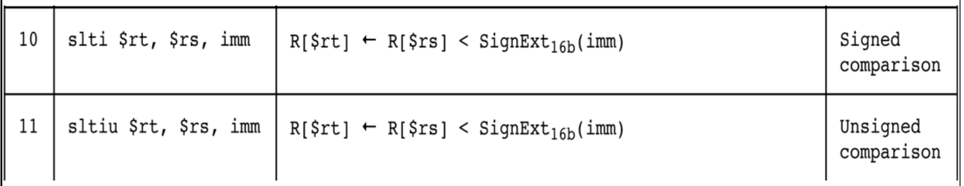
I type Arithmetic and Logical Instructions



08	addi \$rt, \$rs, imm	R[\$rt] ← R[\$rs] + SignExt_{16b}(imm)	Exception on signed overflow
09	addiu \$rt, \$rs, imm	R[\$rt] ← R[\$rs] + SignExt_{16b}(imm)	
12	andi \$rt, \$rs, imm	R[\$rt] ← R[\$rs] & {0 × 16, imm}	
13	ori \$rt, \$rs, imm	R[\$rt] ← R[\$rs] | {0 × 16, imm}	
14	xori \$rt, \$rs, imm	R[\$rt] ← R[\$rs] ^ {0 × 16, imm}	

Note the sign extension Vs zero extension of the immediate operand for arithmetic Vs logical ops

I type comparison ops



10	slti \$rt, \$rs, imm	R[\$rt] ← R[\$rs] < SignExt_{16b}(imm)	Signed comparison
11	sltiu \$rt, \$rs, imm	R[\$rt] ← R[\$rs] < SignExt_{16b}(imm)	Unsigned comparison

Note sign extension for comparison since its an arithmetic operation

We need more space!

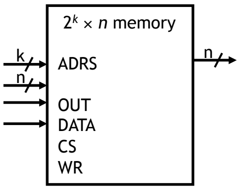
- Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bits wide.
  - That's not enough to hold data structures like large arrays.
  - We also can't access data elements that are wider than 32 bits.

- We need to add some main memory to the system!
  - RAM is cheaper and denser than registers, so we can add lots of it.
  - But memory is also significantly slower, so registers should be used whenever possible.

- In the past, using registers wisely was the programmer's job.
  - For example, C has a keyword "register" that marks commonly-used variables which should be kept in the register file if possible.
  - However, modern compilers do a pretty good job of using registers intelligently and minimizing RAM accesses.

Memory review

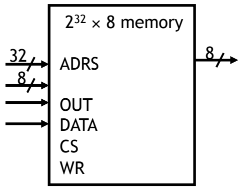
- Memory sizes are specified much like register files; here is a 2^k x *n* RAM.



CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

- A chip select input CS enables or "disables" the RAM.
- ADRS specifies the memory location to access.
- WR selects between reading from or writing to the memory.
  - To read from memory, WR should be set to 0. OUT will be the n-bit value stored at ADRS.
  - To write to memory, we set WR = 1. DATA is the n-bit value to store in memory.

MIPS memory

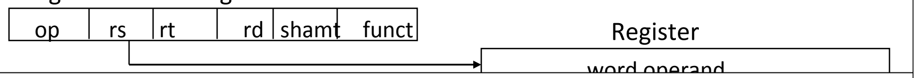


- MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity.
- The MIPS architecture can support up to 32 address lines.
  - This results in a 2^{32} x 8 RAM, which would be 4 GB of memory.
  - Not all actual MIPS machines will have this much!

Address	0	1	2	3	4	5	6	7	8	9	10	11	
8-bit data													○ ○ ○

MIPS Addressing Modes

1. Register addressing



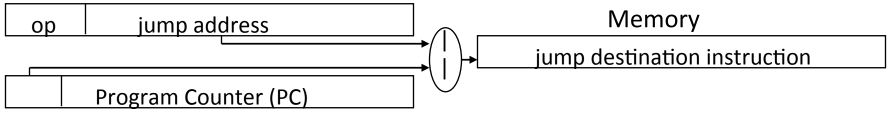
Psuedo Direct Addressing:

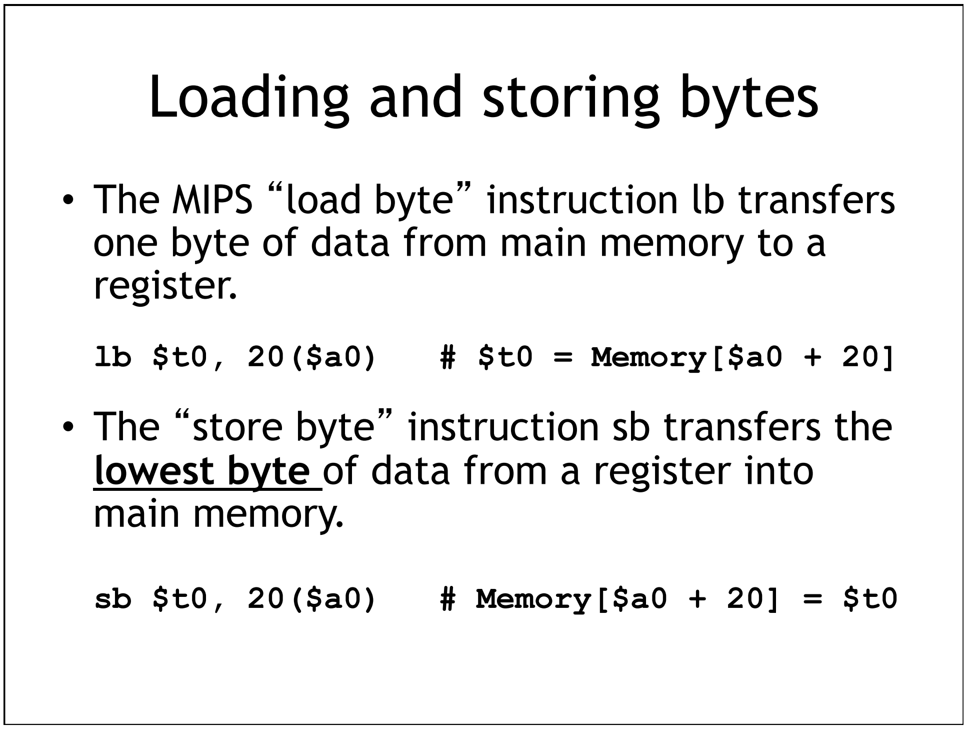
The jump address has only 26 bits – how do you produce a 32 bit address?

4 bits (MS) come from the PC
Next 26 bits come from the jump address
Last 2 bits is 00 always – WHY???

How many possible addresses can you jump to?

5. Pseudo-direct addressing





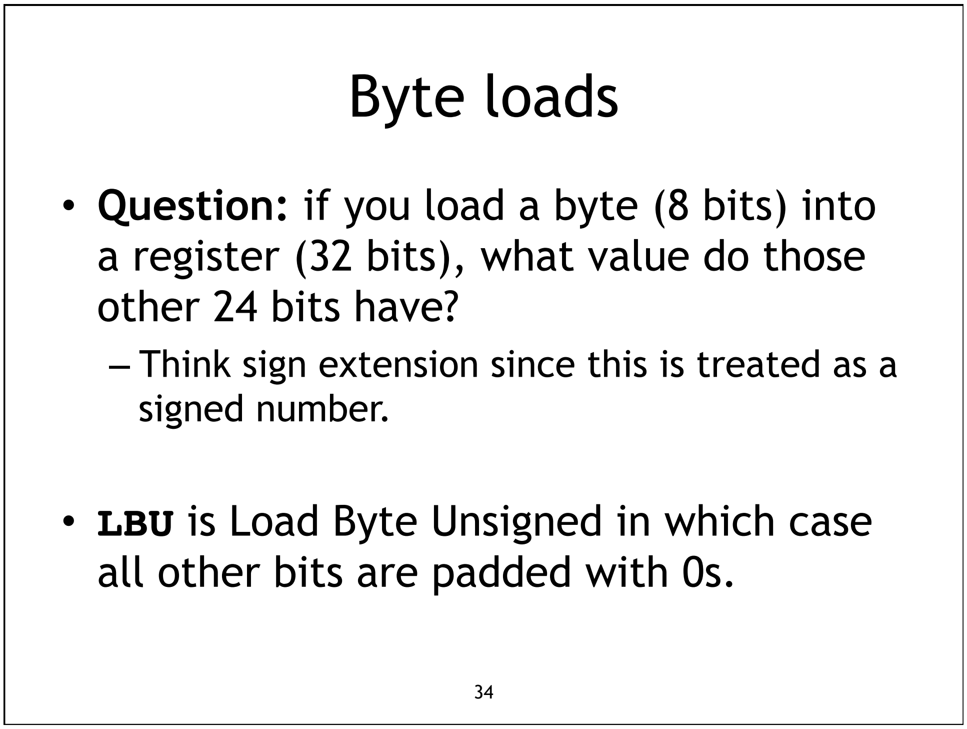
Loading and storing bytes

- The MIPS “load byte” instruction lb transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)   # $t0 = Memory[$a0 + 20]
```

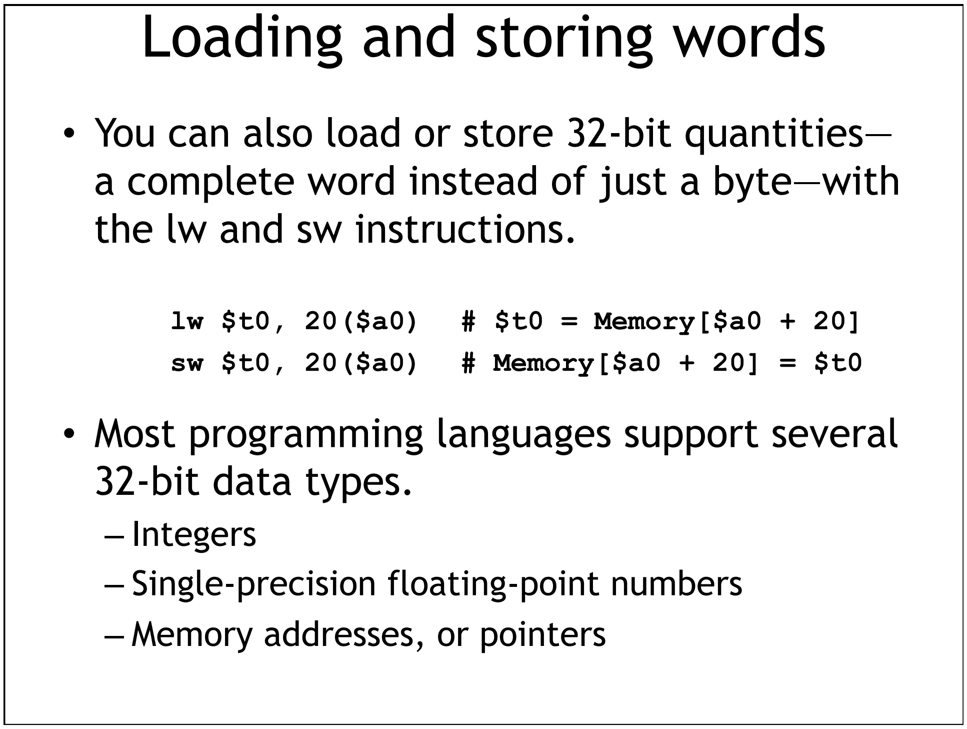
- The “store byte” instruction sb transfers the **lowest byte** of data from a register into main memory.

```
sb $t0, 20($a0)   # Memory[$a0 + 20] = $t0
```



Byte loads

- **Question:** if you load a byte (8 bits) into a register (32 bits), what value do those other 24 bits have?
  - Think sign extension since this is treated as a signed number.
- **`LBU`** is Load Byte Unsigned in which case all other bits are padded with 0s.

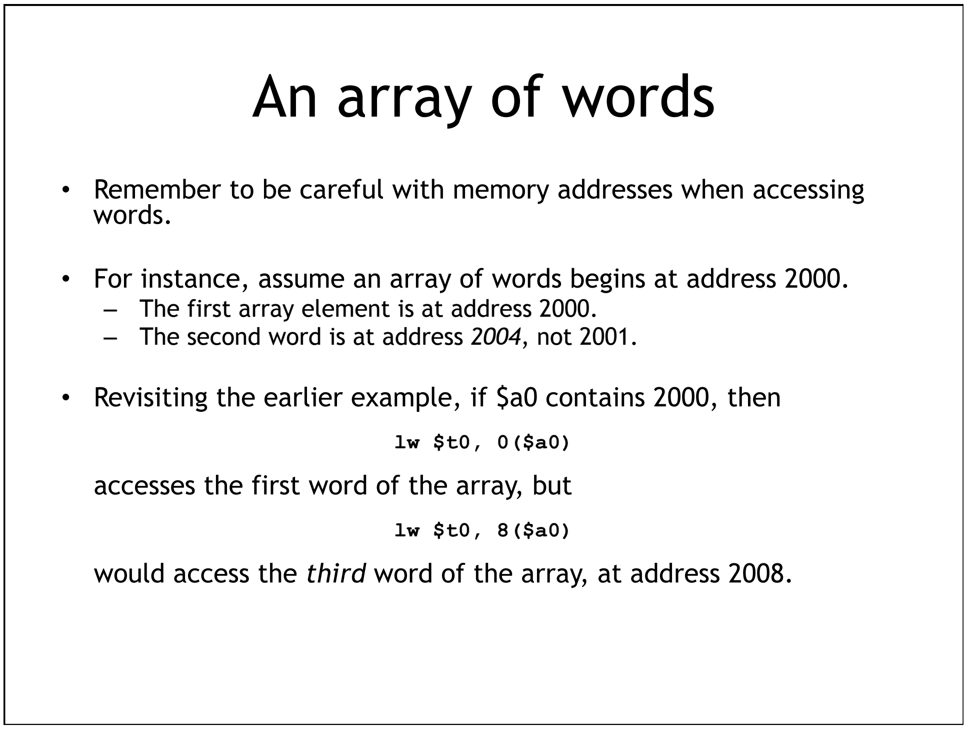


Loading and storing words

- You can also load or store 32-bit quantities—a complete word instead of just a byte—with the lw and sw instructions.

```
lw $t0, 20($a0)   # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)   # Memory[$a0 + 20] = $t0
```

- Most programming languages support several 32-bit data types.
  – Integers
  – Single-precision floating-point numbers
  – Memory addresses, or pointers



An array of words

- Remember to be careful with memory addresses when accessing words.
- For instance, assume an array of words begins at address 2000.
  – The first array element is at address 2000.
  – The second word is at address *2004*, not 2001.
- Revisiting the earlier example, if $a0 contains 2000, then

```
lw $t0, 0($a0)
```

accesses the first word of the array, but

```
lw $t0, 8($a0)
```

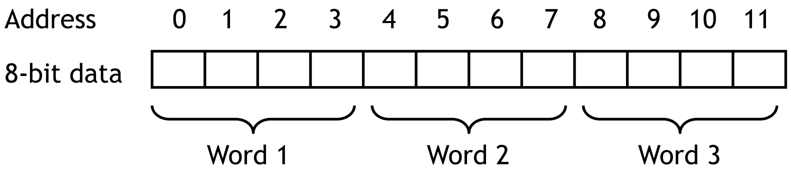
would access the *third* word of the array, at address 2008.

Computing with memory

- So, to compute with memory-based data, you must:
  1. Load the data from memory to the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.

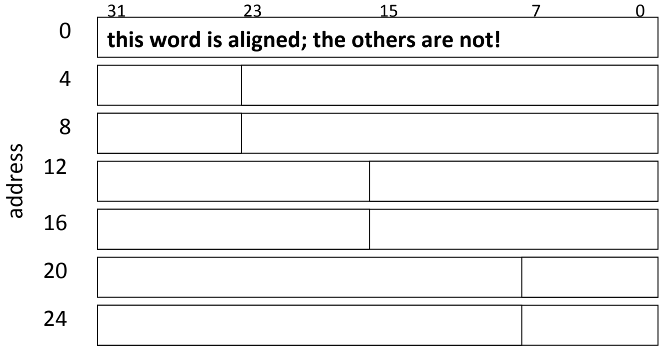
Memory alignment

- Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies **<u>four</u>** contiguous locations (bytes) of main memory.



- The MIPS architecture requires words to be aligned in memory; 32-bit words must start at an address that is divisible by 4.
  - 0, 4, 8 and 12 are valid word addresses.
  - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
  - Unaligned memory accesses result in a bus error.
- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

Memory layout: Alignment

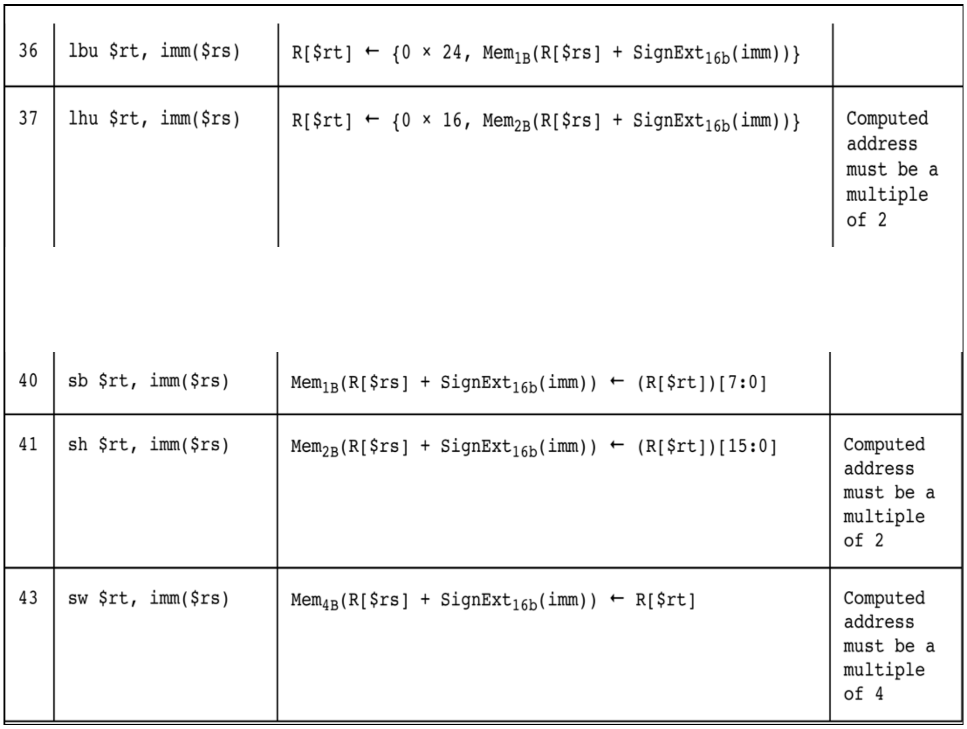


What are the least 2 significant bits of a word **address**?

39

Summary of Memory Instructions (I Type)

15	lui $rt, imm	R[$rt] ← {(imm)[15:0], 0 × 16}	
32	lb $rt, imm($rs)	R[$rt] ← SignExt$_{8b}$(Mem$_{1B}$(R[$rs] + SignExt$_{16b}$(imm)))	
33	lh $rt, imm($rs)	R[$rt] ← SignExt$_{16b}$(Mem$_{2B}$(R[$rs] + SignExt$_{16b}$(imm)))	Computed address must be a multiple of 2
34	lw $rt, imm($rs)	R[$rt] ← Mem$_{4B}$(R[$rs] + SignExt$_{16b}$(imm))	Computed address must be a multiple of 4



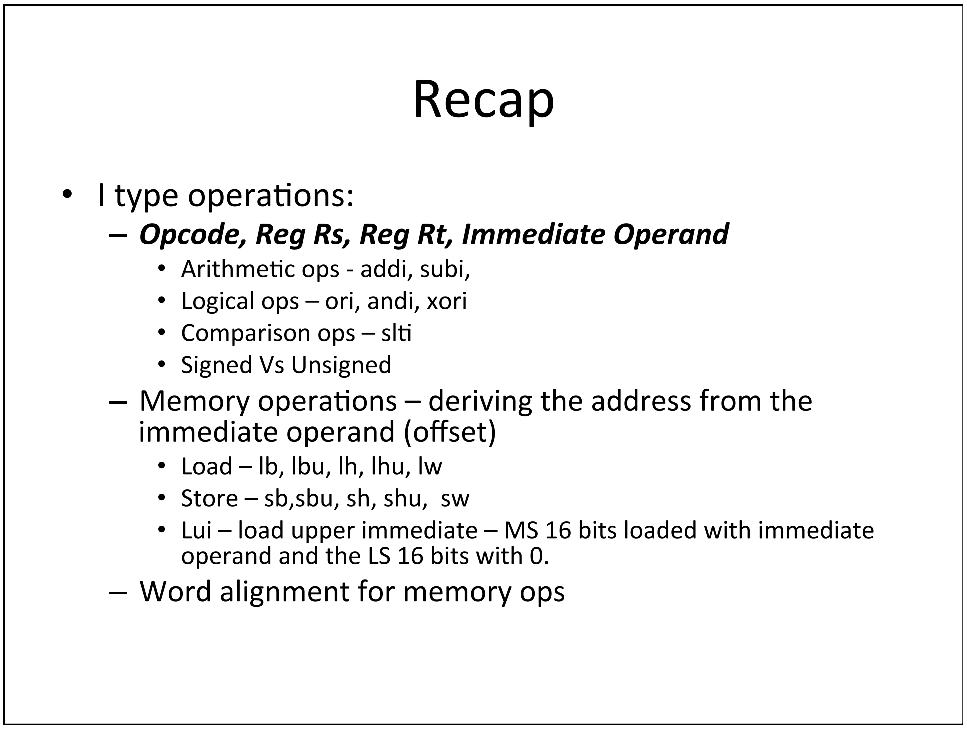
36	lbu $rt, imm($rs)	$R[\$rt] \leftarrow \{0 \times 24, \mathrm{Mem}_{1B}(R[\$rs] + \mathrm{SignExt}_{16b}(imm))\}$	
37	lhu $rt, imm($rs)	$R[\$rt] \leftarrow \{0 \times 16, \mathrm{Mem}_{2B}(R[\$rs] + \mathrm{SignExt}_{16b}(imm))\}$	Computed address must be a multiple of 2
40	sb $rt, imm($rs)	$\mathrm{Mem}_{1B}(R[\$rs] + \mathrm{SignExt}_{16b}(imm)) \leftarrow (R[\$rt])[7:0]$	
41	sh $rt, imm($rs)	$\mathrm{Mem}_{2B}(R[\$rs] + \mathrm{SignExt}_{16b}(imm)) \leftarrow (R[\$rt])[15:0]$	Computed address must be a multiple of 2
43	sw $rt, imm($rs)	$\mathrm{Mem}_{4B}(R[\$rs] + \mathrm{SignExt}_{16b}(imm)) \leftarrow R[\$rt]$	Computed address must be a multiple of 4

Recap

- Three instruction formats for MIPS

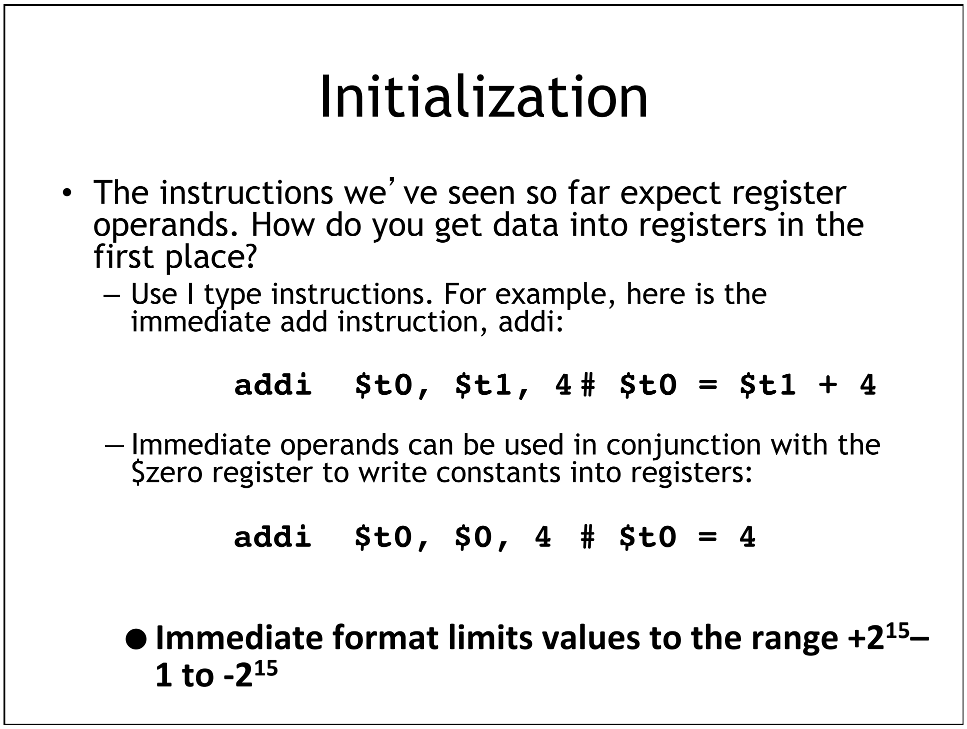
1. R type – Opcode, 3 registers, SHAMT, Func

– Arithmetic ops
– Mul and Div - Use of HI and LO
– Signed Vs Unsigned versions
– Bit operations – shifting (arithmetic Vs Logical), &, |, ExOR
– Comparison – slt, sltu
– Jump <Reg>, Jump and Link <Reg> <Reg>



Recap

- I type operations:
  - ***Opcode, Reg Rs, Reg Rt, Immediate Operand***
    - Arithmetic ops - addi, subi,
    - Logical ops – ori, andi, xori
    - Comparison ops – slti
    - Signed Vs Unsigned
  - Memory operations – deriving the address from the immediate operand (offset)
    - Load – lb, lbu, lh, lhu, lw
    - Store – sb,sbu, sh, shu, sw
    - Lui – load upper immediate – MS 16 bits loaded with immediate operand and the LS 16 bits with 0.
  - Word alignment for memory ops



Initialization

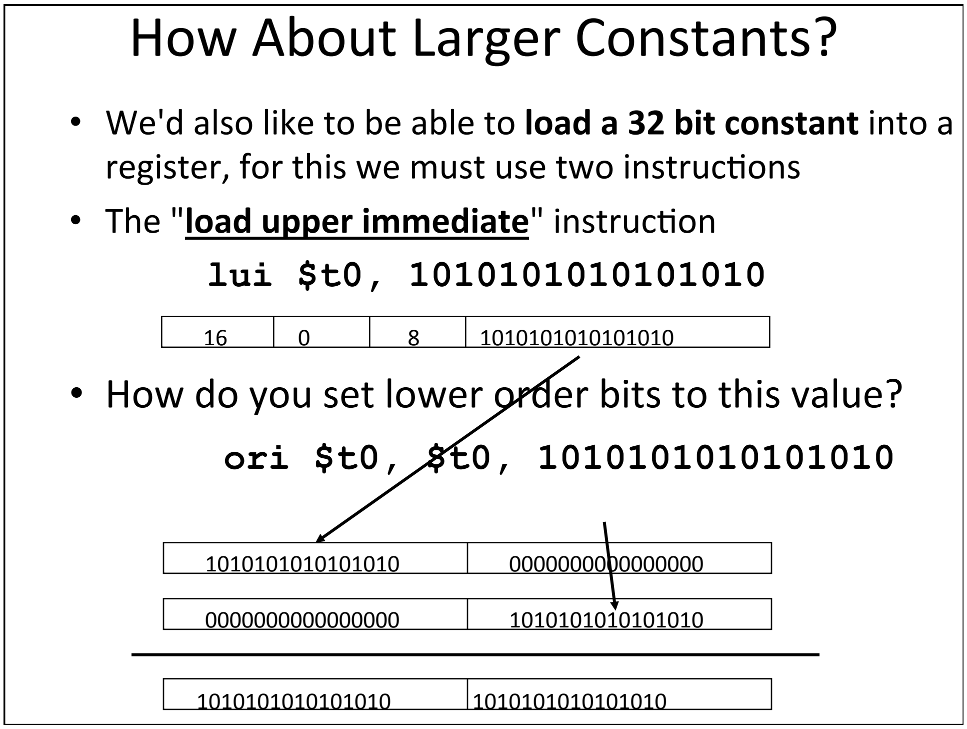
- The instructions we’ ve seen so far expect register operands. How do you get data into registers in the first place?
  - Use I type instructions. For example, here is the immediate add instruction, addi:

```
addi  $t0, $t1, 4# $t0 = $t1 + 4
```

  - Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi  $t0, $0, 4 # $t0 = 4
```

- **Immediate format limits values to the range +2^{15}– 1 to -2^{15}**



How About Larger Constants?

- We'd also like to be able to **load a 32 bit constant** into a register, for this we must use two instructions
- The "**<u>load upper immediate</u>**" instruction

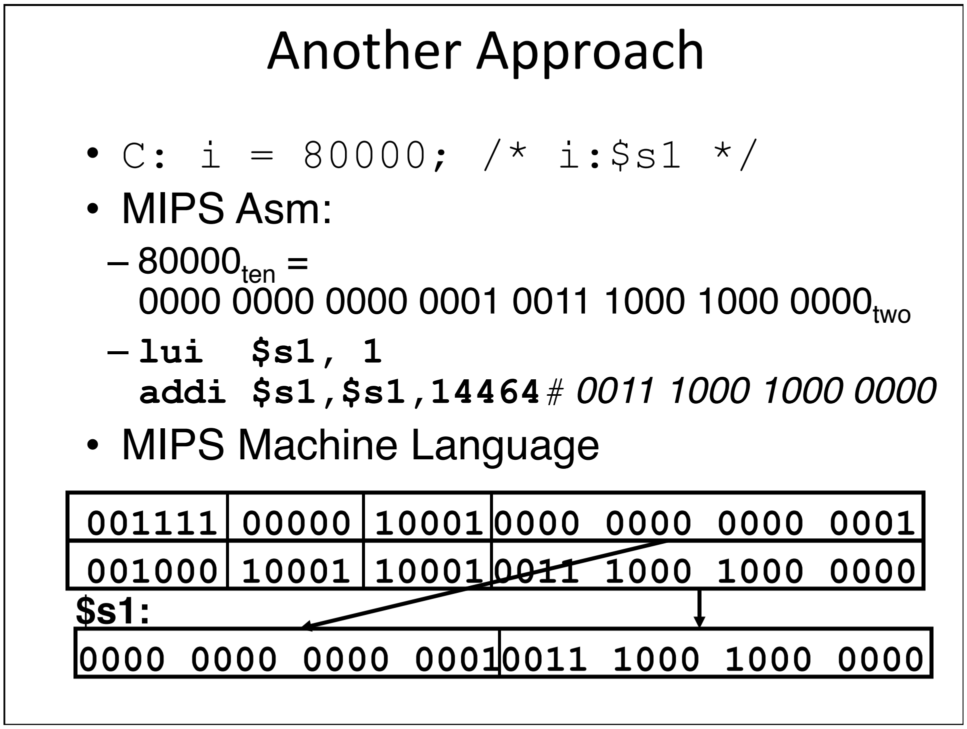
```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010

- How do you set lower order bits to this value?

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010



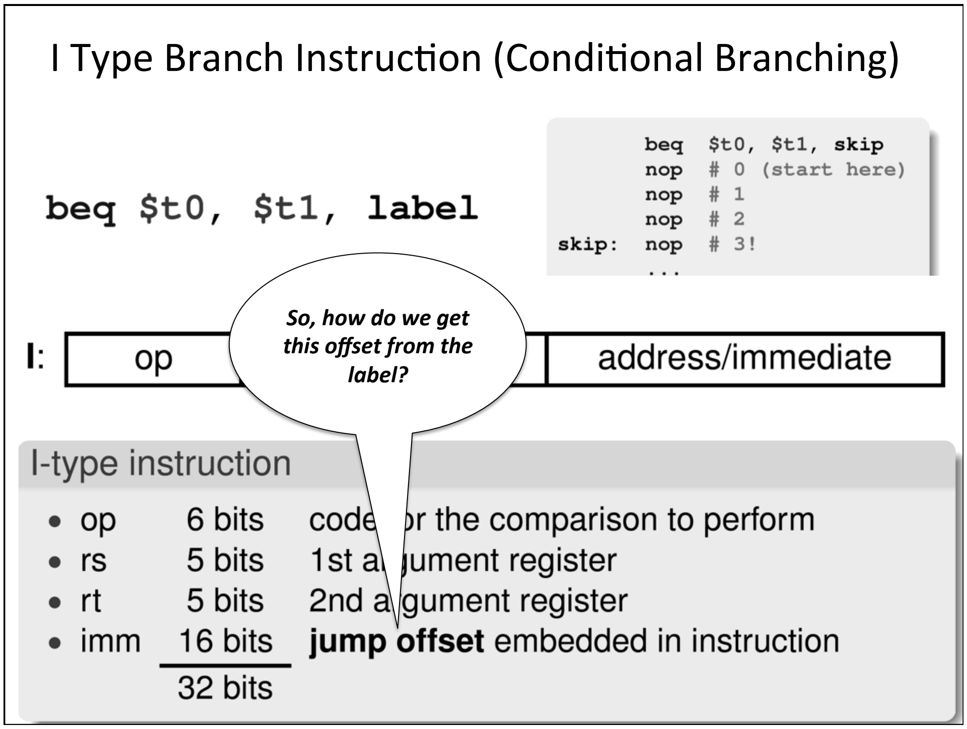
Another Approach

- `C: i = 80000; /* i:$s1 */`
- MIPS Asm:
  - 80000_{ten} = 0000 0000 0000 0001 0011 1000 1000 0000_{two}
  - **`lui  $s1, 1`**
    `addi $s1,$s1,14464` *# 0011 1000 1000 0000*
- MIPS Machine Language

001111	00000	10001	0000 0000 0000 0001
001000	10001	10001	0011 1000 1000 0000

$s1:

0000 0000 0000 0001	0011 1000 1000 0000



I Type Branch Instruction (Conditional Branching)

```
beq $t0, $t1, label
```

```
       beq  $t0, $t1, skip
       nop  # 0 (start here)
       nop  # 1
       nop  # 2
skip:  nop  # 3!
       ...
```

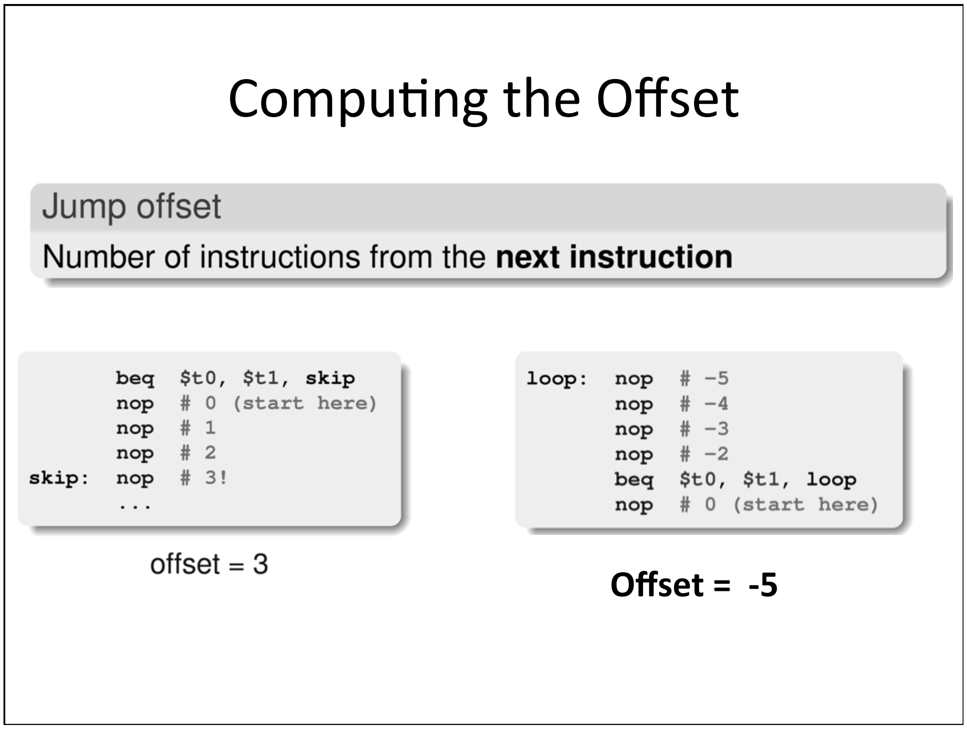
I: op | address/immediate

So, how do we get this offset from the label?

I-type instruction

- op 6 bits code or the comparison to perform
- rs 5 bits 1st argument register
- rt 5 bits 2nd argument register
- imm 16 bits **jump offset** embedded in instruction

32 bits



Computing the Offset

Jump offset

Number of instructions from the **next instruction**

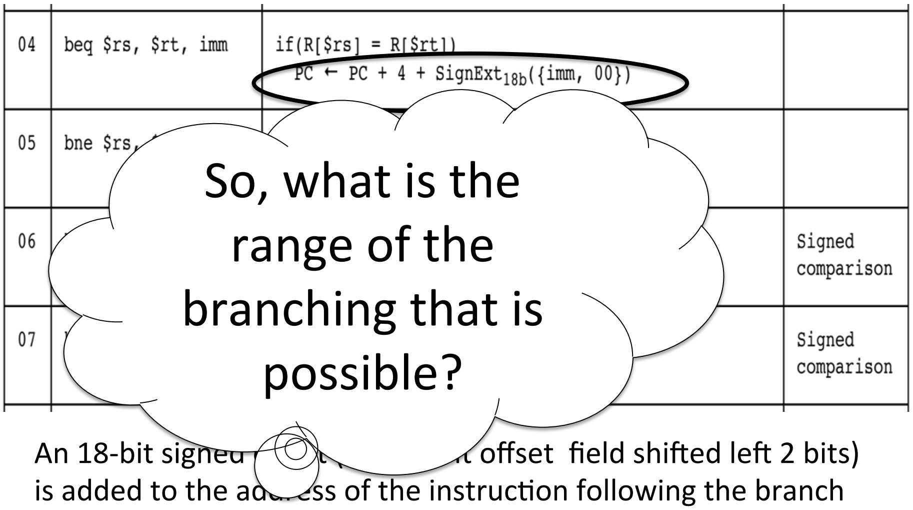
```
       beq  $t0, $t1, skip
       nop  # 0 (start here)
       nop  # 1
       nop  # 2
skip:  nop  # 3!
       ...
```

offset = 3

```
loop:  nop  # -5
       nop  # -4
       nop  # -3
       nop  # -2
       beq  $t0, $t1, loop
       nop  # 0 (start here)
```

Offset = -5

I Type Branch Instructions



04	beq $rs, $rt, imm	if(R[$rs] = R[$rt])
		PC ← PC + 4 + SignExt$_{18b}$({imm, 00})	

05	bne $rs,		
06			Signed comparison
07			Signed comparison

So, what is the range of the branching that is possible?

An 18-bit signed t offset field shifted left 2 bits) is added to the ad ss of the instruction following the branch (not the branch itself), to form a PC-relative effective target address.

J Type Instructions

J:	op	target address

Relative vs. absolute addressing

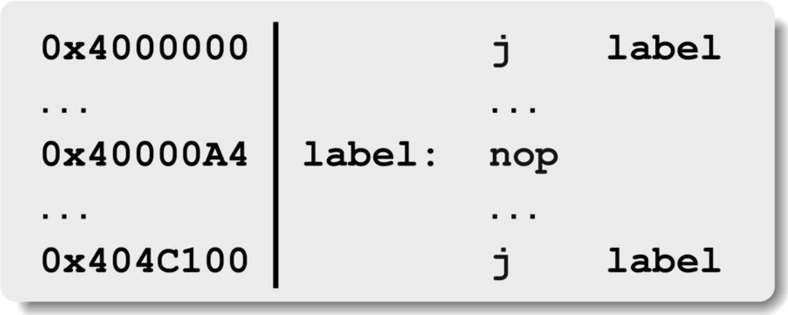
Branch instructions – offset is relative:

PC = PC + 4 + offset × 4

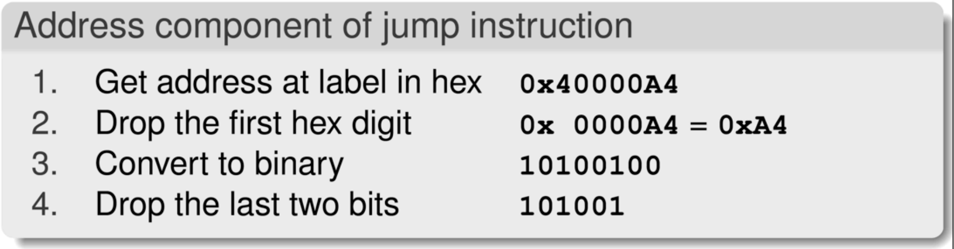
Jump instructions – address is absolute:

PC = (PC & 0xF0000000) | (address × 4)

Address of the Jump



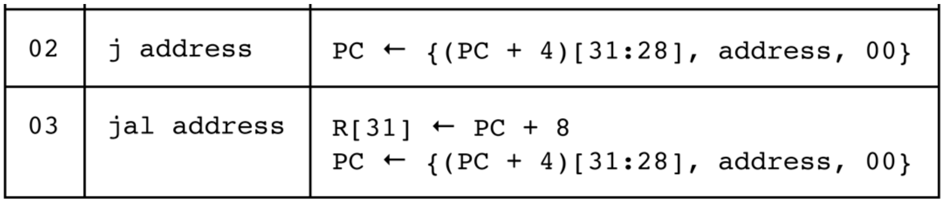
```
0x4000000            j    label
...                  ...
0x40000A4   label:   nop
...                  ...
0x404C100            j    label
```



Address component of jump instruction

1.	Get address at label in hex	**0x40000A4**
2.	Drop the first hex digit	**0x 0000A4 = 0xA4**
3.	Convert to binary	**10100100**
4.	Drop the last two bits	**101001**

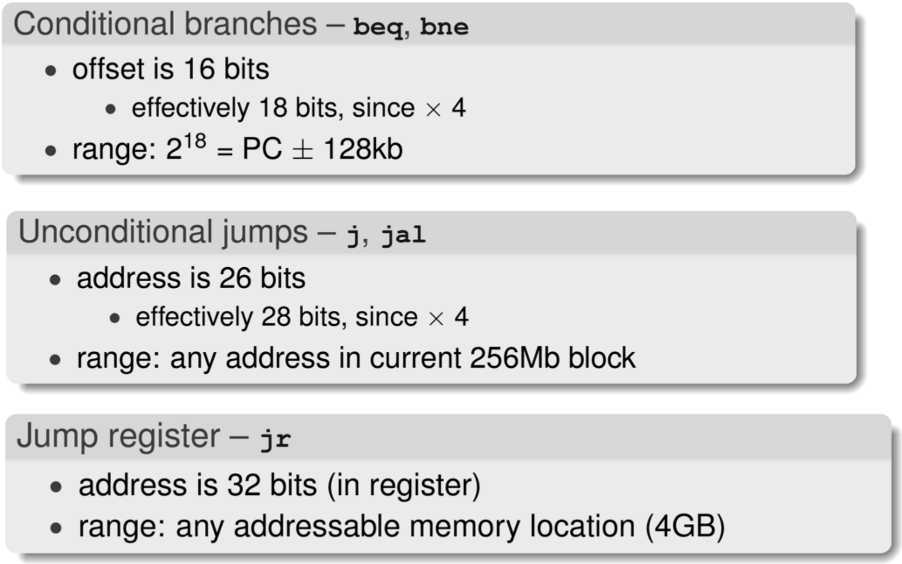
J Type Instructions



02	j address	PC ← {(PC + 4)[31:28], address, 00}

03	jal address	R[31] ← PC + 8
		PC ← {(PC + 4)[31:28], address, 00}

Branch Vs Jump



Conditional branches – **beq**, **bne**

- offset is 16 bits
  - effectively 18 bits, since × 4
- range: 2^{18} = PC ± 128kb

Unconditional jumps – **j**, **jal**

- address is 26 bits
  - effectively 28 bits, since × 4
- range: any address in current 256Mb block

Jump register – **jr**

- address is 32 bits (in register)
- range: any addressable memory location (4GB)

Pseudo Instructions

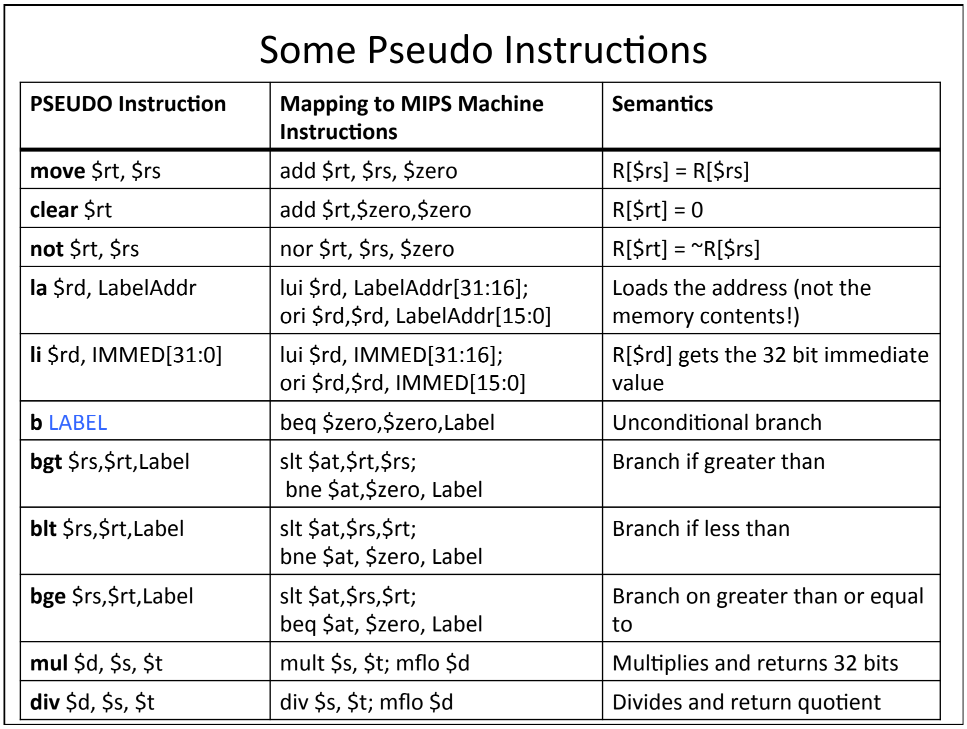
- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, “real” instructions.
- Examples: the **li** and **move** pseudo-instructions:

```
li $a0, 2000# Load immediate 2000 into $a0
Move $a1, $t0# Copy $t0 into $a1
```

- They are probably clearer than their corresponding MIPS instructions:

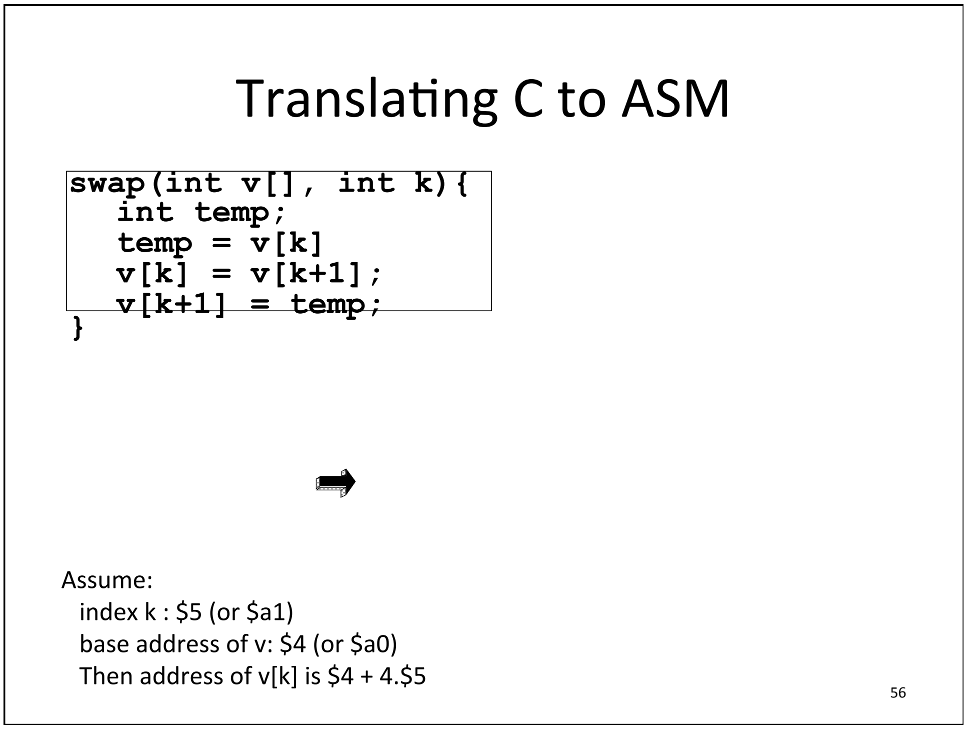
```
addi $a0, $0, 2000# Initialize $a0 to 2000
Add  $a1, $t0, $0 # Copy $t0 into $a1
```

- We’ll see lots more pseudo-instructions this semester. A complete list of instructions is given in your text and on the reference that will be posted.
  – Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.



Some Pseudo Instructions

PSEUDO Instruction	**Mapping to MIPS Machine Instructions**	**Semantics**
move $rt, $rs	add $rt, $rs, $zero	R[$rs] = R[$rs]
clear $rt	add $rt,$zero,$zero	R[$rt] = 0
not $rt, $rs	nor $rt, $rs, $zero	R[$rt] = ~R[$rs]
la $rd, LabelAddr	lui $rd, LabelAddr[31:16]; ori $rd,$rd, LabelAddr[15:0]	Loads the address (not the memory contents!)
li $rd, IMMED[31:0]	lui $rd, IMMED[31:16]; ori $rd,$rd, IMMED[15:0]	R[$rd] gets the 32 bit immediate value
b LABEL	beq $zero,$zero,Label	Unconditional branch
bgt $rs,$rt,Label	slt $at,$rt,$rs; bne $at,$zero, Label	Branch if greater than
blt $rs,$rt,Label	slt $at,$rs,$rt; bne $at, $zero, Label	Branch if less than
bge $rs,$rt,Label	slt $at,$rs,$rt; beq $at, $zero, Label	Branch on greater than or equal to
mul $d, $s, $t	mult $s, $t; mflo $d	Multiplies and returns 32 bits
div $d, $s, $t	div $s, $t; mflo $d	Divides and return quotient



Translating C to ASM

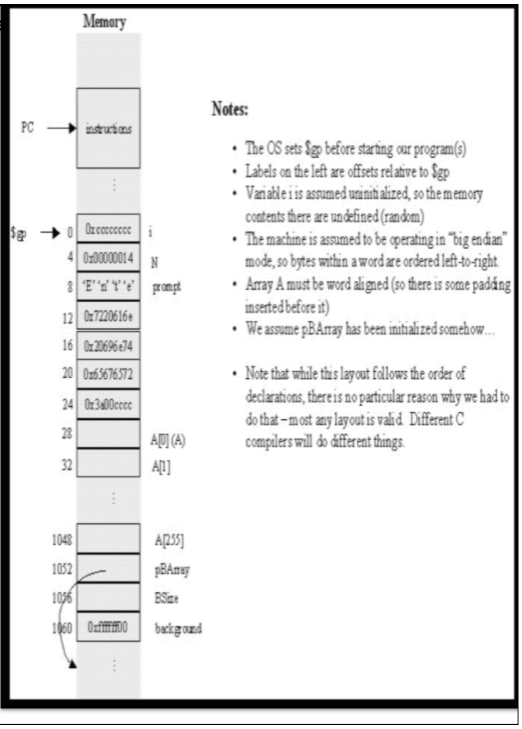
```
swap(int v[], int k){
   int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

➡

Assume:
  index k : $5 (or $a1)
  base address of v: $4 (or $a0)
  Then address of v[k] is $4 + 4.$5

```
// none of these allocate storage
   #define MAX_SIZE  256
   #define IF(a)    if (a) {
   #define ENDIF    }
   typedef struct {
       unsigned char red;
      unsigned char green;
       unsigned char blue;
       unsigned char alpha;
   } RGBa;

   // these allocate storage
   int     i;
   int     N = 20;
   char    prompt[] =
      "Enter an integer:";
   int     A[MAX_SIZE];
   int*    pBArray;
   int     BSize;
   RGBa    background =
      {0xff, 0xff, 0xff, 0x0};
```



Notes:

- The OS sets $gp before starting our program(s)
- Labels on the left are offsets relative to $gp
- Variable i is assumed uninitialized, so the memory contents there are undefined (random)
- The machine is assumed to be operating in "big endian" mode, so bytes within a word are ordered left-to-right.
- Array A must be word aligned (so there is some padding inserted before it)
- We assume pBArray has been initialized somehow…

- Note that while this layout follows the order of declarations, there is no particular reason why we had to do that – most any layout is valid. Different C compilers will do different things.

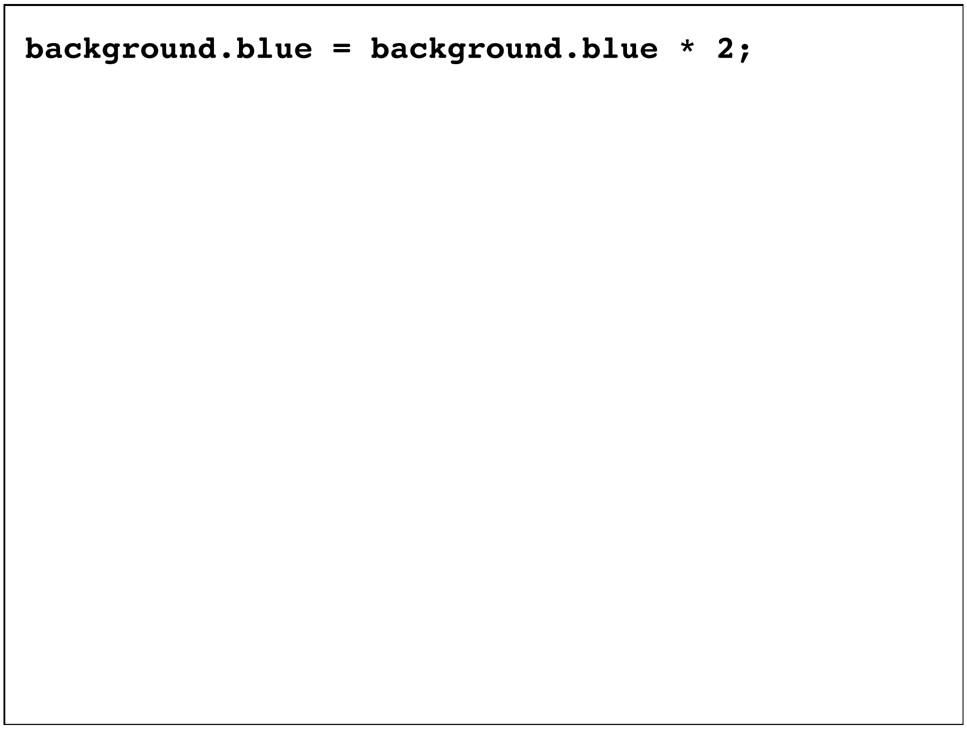
```
int     i;
int     N = 20;
i = N*N + 3*N
```

```
A[i] = A[i/2] + 1;

A[i+i] = -1;
```

```
// set N to the smallest odd no less than N
    if ( N%2 == 0 ) N++;
```

Can you translate to ASM without branching?



```
background.blue = background.blue * 2;
```