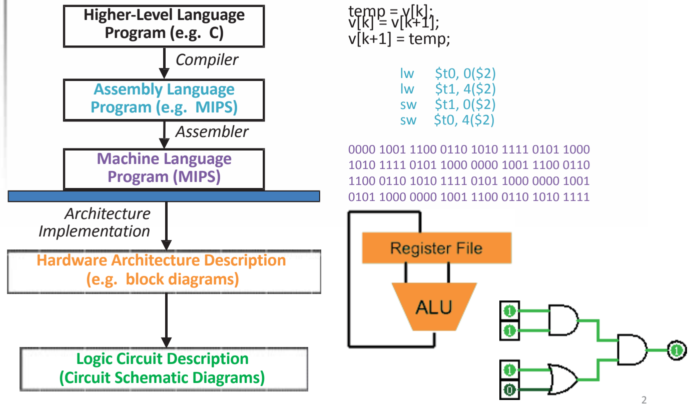
MIPS Instruction Set Architecture

Idea #1: Levels of Representation


Higher-Level Language Program (e.g. C)
Compiler
Assembly Language Program (e.g. MIPS)
Assembler
Machine Language Program (MIPS)
Architecture Implementation
Hardware Architecture Description (e.g. block diagrams)
Logic Circuit Description (Circuit Schematic Diagrams)
Register File
ALU

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Why Study Assembly?

- Understand computers at a deeper level
  - Learn to write more compact and efficient code
  - Can sometimes hand optimize better than a compiler
- More sensible for minimalistic applications
  - e.g. distributed sensing and systems
  - Eliminating OS, compilers, etc. reduce size and power consumption
  - Embedded computers outnumber PCs!

Recall: Reduced Instruction Set Computing

- The early trend was to add more and more instructions to do elaborate operations this became known as *Complex Instruction Set Computing*(CISC)
- Opposite philosophy later began to dominate: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

Common RISC Simplifications

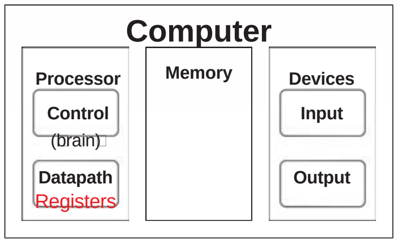
- **Fixed instruction length:**
  Simplifies fetching instructions from memory
- **Simplified addressing modes:**
  Simplifies fetching operands from memory
- **Few and simple instructions in the instruction set:**
  Simplifies instruction execution
- **Minimize memory access instructions (load/store):**
  Simplifies necessary hardware for memory access
- **Let compiler do heavy lifting:**
  Breaks complex statements into multiple assembly instructions

Mainstream ISAs

- Intel 80x86
  - Used in Macbooks and PCs
  - Found in Core i3, Core i5, Core i7, etc.
- Advanced RISC Machine (ARM)
  - Smart phone-like devices: iPhone, iPad, iPod, etc.
  - The most popular RISC (20x more common than 80x86)
- MIPS
  - Networking equipment, PS2, PSP
  - Very similar to ARM

Five Components of a Computer

- We begin our study of how a computer works!
  - Control
  - Datapath
  - Memory
  - Input
  - Output



- Registers are part of the Datapath

Recall: Computer Hardware Operands

- In high-level languages, number of variables limited only by available memory
- ISAs have a fixed, small number of operands called registers
  - Special locations built directly into hardware
  - **Benefit:**Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  - **Drawback:**Operations can only be performed on these predetermined number of registers

MIPS Registers (1/2)

- MIPS has 32 general-purpose registers (R0-31)
  - Each register is 32 bits wide and holds a word
- Tradeoff between speed and availability
  - Smaller number means faster hardware but insufficient to hold data for typical C programs
- *Registers have no type* (C concept); the operation being performed determines how register contents are treated
- MIPS has also a separate set of 32 floating-point registers (F0-31)

MIPS Registers (2/2)

- Register denoted by $ can be referenced by number (`$0`-`$31`) or name:
  - Registers that hold programmer variables:

    `$s0-$s7` $\longleftrightarrow$ **`$16-$23`**
  - Registers that hold temporary variables:

    `$t0-$t7` $\longleftrightarrow$ **`$8-$15`**

    `$t8-$t9` $\longleftrightarrow$ **`$24-$25`**
  - You'll learn about the other 14 registers later
- In general, using register names makes code more readable

MIPS Instructions (1/2)

- Instruction Syntax is rigid:

  `op dst, src1, src2`

  - 1 operator, 3 operands
    - **`op`**= operation name (operator)
    - **`dst`**= register getting result (destination)
    - **`src1`**= first register for operation (source 1)
    - **`src2`**= second register for operation (source 2)
- Keep hardware simple via regularity

MIPS Instructions (2/2)

- One operation per instruction, at most one instruction per line
- Assembly instructions are related to C operations (`=`, `+`, `-`, `*`, `/`, `&`, `|`, etc.)
  - Must be, since C code decomposes into assembly!
  - A single line of C may break up into several lines of MIPS

MIPS Instructions Example

- Your very first instructions!
  (assume here that the variables **a, b,** and **c**are assigned to registers **$s1, $s2,** and **$s3**, respectively)
- Integer Addition (**add**)
  - C:`a = b + c`
  - MIPS: `add  $s1, $s2, $s3`
- Integer Subtraction (**sub**)
  - C:`a = b - c`
  - MIPS:`sub  $s1, $s2, $s3`

MIPS Instructions Example

- Suppose a→$s0,b→$s1,c→$s2, d→$s3, and e→$s4. Convert the following C statement to MIPS:

```
a = (b + c) – (d + e);
```

```
add $t1, $s3, $s4
add $t2, $s1, $s2
sub $s0, $t2, $t1
```

Ordering of instructions matters (must follow order of operations)

Utilize temporary registers

Comments in MIPS

- Comments in MIPS follow hash mark (#) until the end of line
  - Improves readability and helps you keep track of variables/registers!

```
add $t1, $s3, $s4 # $t1=d+e
add $t2, $s1, $s2 # $t2=b+c
sub $s0, $t2, $t1 # a=(b+c)-(d+e)
```

The Zero Register

- Zero appears so often in code and is so useful that it has its own register!
- Register zero (`$0` or `$zero`) always has the value 0 and cannot be changed!
  - i.e. any instruction with **`$0`** as **`dst`** has no effect
- Example Uses:
  - `add  $s3,  $0,   $0  # c=0`
  - `add  $s1, $s2,   $0  # a=b`

Machine Language: R-type Format

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t1, $s1, $s2
  - registers have numbers, $t1=9, $s1=17, $s2=18
- MIPS R-type instruction format:

000000	10001	10010	01001	00000	100000
op	rs	rt	rd	shamt	funct

Immediates

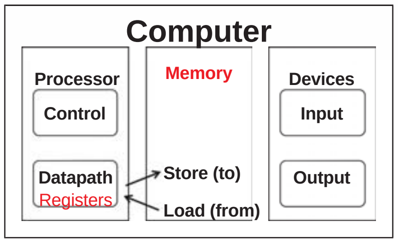
- Numerical constants are called immediates
- Separate instruction syntax for immediates:

  opi dst, src, imm

  - Operation names end with **i**, replace 2nd source register with an immediate (unsigned ops that have **u** at end of opi names)
- Example Uses:
  - **`addi $s1, $s2, 5  # a=b+5`**
  - **`addi $s3, $s3, 1  # c++`**
- Why no **subi** instruction?

Five Components of a Computer

- Data Transfer instructions are between registers (Datapath) and Memory
  - Allow us to fetch and store operands in memory



Data Transfer

- C variables map onto **registers**;
  What about large data structures like arrays?
  - Dont forget ***memory***, our one-dimensional array indexed by byte addresses starting at 0
- MIPS instructions only operate on registers!
- Specialized data transfer instructions move data between registers and memory
  - Store: register TO memory
  - Load: register FROM memory

Data Transfer

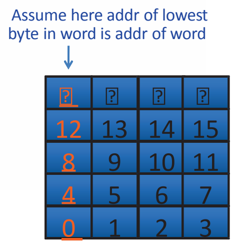
- Instruction syntax for data transfer:

  `op reg, off(baseAddr_reg)`

  - **`op`**= operation name (operator)
  - **`reg`**= register for operation source or destination
  - **`baseAddr_reg`**= register with pointer to memory (base address)
  - **`off`**= address offset (immediate) in bytes (offset aka displacement)
- Accesses memory at address **`Reg[bAddr_reg]+off`**
- **Reminder:** A register holds a word of raw data (no type) make sure to use a register (and offset) that point to a valid memory address

Memory is Byte-Addressed

- What is the smallest data type in C?
  - char, which was a *byte* (8 bits)
  - Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)
- Memory addresses are indexed by ***bytes***, not words
- Word addresses are 4 bytes apart
  - Word addr is same as left-most byte
  - Addrs must be multiples of 4 to be word-aligned
- Pointer arithmetic not done for you in assembly
  - Must take data size into account yourself



Data Transfer Instructions

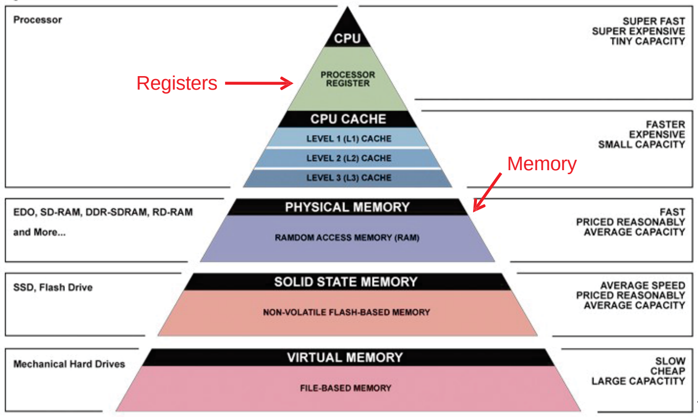
- Load Word (`lw`)
  - Takes data at address `Reg[bAddr_reg]+off` FROM memory and places it into `reg`
- Store Word (`sw`)
  - Takes data in `reg` and stores it TO memory at address `Reg[bAddr_reg]+off`
- Example Usage:

```
# addr of int A[] -> $s3, a -> $s0
lw  $t0,12($s3) # $t0=A[3]
add $t0,$s2,$t0 # $t0=A[3]+a
sw  $t0,40($s3) # A[10]=A[3]+a
```

Registers vs. Memory

- What if more variables than registers?
  - Keep most frequently used in registers and move the rest to memory (called *spilling* to memory)
- Why not all variables in memory?
  - Smaller is faster: registers 100-500 times faster
  - Registers more versatile
    - In 1 arithmetic instruction: read 2 operands, perform 1 operation, and 1 write
    - In 1 data transfer instruction: 1 read/write, no operation

Idea #3: Principle of Locality/ Memory Hierarchy



Machine Language: I-type Format

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: **Good design demands a compromise**
- Introduce a new type of instruction format
  - **I-type** for **data transfer** instructions
  - other format was R-type for register
- Example: **lw $t0, 32($s2)**

35	18	8	32
op	rs	rt	16 bit number

- Where's the compromise?

Constants

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

  e.g., A = A + 5;
  B = B + 1;
  C = C -
  18;

- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like **$zero**) for constants like one.

- MIPS instructions using constants (immediate operands):

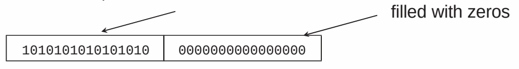
  addi $29, $29,
  4 slti $8, $18,
  10 andi $29,
  $29, 6 ori $29,
  $29, 4

- **Design Principle: Make the common case fast.** *Which format?*

How about larger constants?

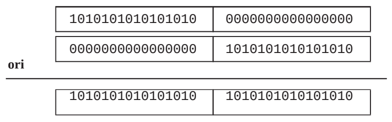
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

lui $t0, 1010101010101010



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

ori $t0, $t0, 1010101010101010



MIPS Signed vs. Unsigned

- MIPS terms signed and unsigned appear in 3 different contexts:
  - Signed vs. unsigned bit extension
    - `lb`
    - `lbu`
  - Detect vs. dont detect overflow
    - `add, addi, sub, mult, div`
    - `addu, addiu, subu, multu, divu`
  - Signed vs. unsigned comparison
    - `slt, slti`
    - `sltu, sltiu`

Chars and Strings

- **Recall:** A string is just an array of characters and a **`char`** in C uses 8-bit ASCII
- Method 1: Move words in and out of memory using bit-masking and shifting

```
lw   $s0,0($s1)
andi $s0,$s0,0xFF # lowest byte
```

- Method 2: Load/store byte instructions

```
lb   $s0,0($s1)
sb   $s0,1($s1)
```

Addrs no longer need to be multiples of 4

Byte Instructions

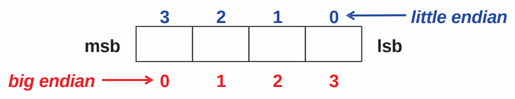
- **`lb/sb`** utilize the least significant byte of the register
  - On **`sb`**, upper 24 bits are ignored
  - On **`lb`**, upper 24 bits are filled by sign-extension
- For example, let **`*($s0) = 0x00000180`**:

```
lb $s1,1($s0)  # $s1=0x00000001
lb $s2,0($s0)  # $s2=0xFFFFFF80
sb $s2,2($s0)  # *($s0)=0x00800180
```

- Normally you dont□want to sign-extend chars
  - Use **`lbu`** (load byte unsigned)

Recall: Endianness

- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte
- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte



- MIPS is big-endian

Computer Decision Making

- In C, outcomes of comparative/logical statements determined which blocks of code to execute
- In MIPS, we cant□define blocks of code; all we have are labels
  - Defined by text followed by a colon (e.g. `main:`) and refers to the instruction that follows
  - Generate control flow by jumping to labels
  - C has these too, but they are considered bad style

Decision Making Instructions

- Branch If Equal (**`beq`**)
  - **`beq reg1,reg2,label`**
  - If value in **`reg1`** = value in **`reg2`**, go to **`label`**
- Branch If Not Equal (**`bne`**)
  - **`bne reg1,reg2,label`**
  - If value in **`reg1`** ≠ value in **`reg2`**, go to **`label`**
- Jump (**`j`**)
  - **`j label`**
  - Unconditional jump to **`label`**

Instruction Formats for Branches & Jumps

- Instructions:

bne $t4,$t5,Label	Next instruction is at Label if $t4 ≠$t5
beq $t4,$t5,Label	Next instruction is at Label if $t4 = $t5
j Label	Next instruction is at Label

- Formats:

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

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

Addresses in Branches and Jumps

- Instructions:

bne $t4,$t5,Label	Next instruction is at Label if **$t4≠$t5**
beq $t4,$t5,Label	Next instruction is at Label if **$t4=$t5**

- Formats:

I	op	rs	rt	16 bit address

Could specify a register (like lwand sw) and add it to address

  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

Breaking Down the If Else

C Code:

```
if(i==j) {
  a = b  /* then */
} else {
  a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

MIPS (beq):

```
# i→$s0, j→$s1
# a→$s2, b→$s3

beq $s0,$s1,???
???     ⟵ This label unnecessary
sub $s2, $0, $s3
j   end
then:
add $s2, $s3, $0
end:
```

Breaking Down the If Else

C Code:

```
if(i==j) {
  a = b  /* then */
} else {
  a = a-b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

MIPS (bne):

```
# i→$s0, j→$s1
# a→$s2, b→$s3
bne $s0,$s1,else

add $s2, $s3, $0
j   end
else:
sub $s2, $s2, $s3
end:
```

Loops in MIPS

- There are three types of loops in C:
  - **`while`**, **`do…while`** , and **`for`**
  - Each can be rewritten as either of the other two, so the same concepts of decision-making apply
- **Key Concept:** Though there are multiple ways to write a loop in MIPS, the key to decision-making is the conditional branch

---

C to MIPS Practice

- Let's put our all of our new MIPS knowledge to use in an example: "Fast String Copy"
- C code is as follows:

```
/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != '\0') ;
```

- What do we know about its structure?
  - Single **`while`** loop
  - Exit condition is an equality test

C to MIPS Practice

- Start with code skeleton:

```
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop:                       # $t0 = *p
                            # *q = $t0
                            # p = p + 1
                            # q = q + 1
                            # if *p==0, go to Exit
       j Loop               # go to Loop
Exit:
```

C to MIPS Practice

Finished code :

```
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop: lb   $t0,0($s0)  # $t0 = *p
      sb   $t0,0($s1)  # *q = $t0
      addi $s0,$s0,1   # p = p + 1
      addi $s1,$s1,1   # q = q + 1
      beq  $t0,$0,Exit # if *p==0, go to Exit
      j Loop           # go to Loop
Exit: # N chars in p => N*6 instructions
```

C to MIPS Practice

- Alternate code using bne:

```
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop: lb   $t0,0($s0)  # $t0 = *p
      sb   $t0,0($s1)  # *q = $t0
      addi $s0,$s0,1   # p = p + 1
      addi $s1,$s1,1   # q = q + 1
      bne  $t0,$0,Loop # if *p!=0, go to Loop
# N chars in p => N*5 instructions
```

Other MIPS Arithmetic Instructions

- The following commands place results in the special registers **HI** and **LO**
  - Access these values with "move from HI" (**mfhi dst**) and "move from LO" (**mflo dst**)
- Multiplication (**mult**)
  - **mult src1,src2**
  - **src1*src2**: lower 32-bits in **LO**, upper 32-bits in **HI**
- Division (**div**)
  - **div src1,src2**
  - **src1/src2:**puts quotient in **LO**, remainder in **HI**

MIPS Arithmetic Instructions

- Example:

```
# mod using div: $s2 = $s0 mod $s1
mod:
div  $s0,$s1 # LO = $s0/$s1
mfhi $s2     # HI = $s0 mod $s1
```

Arithmetic Overflow

- **Recall:** **Overflow** occurs when there is a mistake in arithmetic due to the limited precision in computers
  - i.e. not enough bits to represent answer
- MIPS detects overflow (*throws error*)
  - Arithmetic "unsigned" instructions ignore overflow

Overflow Detection	No Overflow Detection
`add  dst,src1,src2`	`addu  dst,src1,src2`
`addi dst,src1,src2`	`addiu dst,src1,src2`
`sub  dst,src1,src2`	`subu  dst,src1,src2`

Arithmetic Overflow

- Example:

Recall: this is the most negative number

```
# $s0=0x80000000, $s1=0x1
add   $t0,$s0,$s0 # overflow (error)
addu  $t1,$s0,$s0 # $t1=0
addi  $t2,$s0,-1  # overflow (error)
addiu $t2,$s0,-1  # $t3=0x7FFFFFFF
sub   $t4,$s0,$s1 # overflow (error)
subu  $t5,$s0,$s1 # $t5=0x7FFFFFFF
```

MIPS Bitwise Instructions

Note:a→$s1, b→$s2, c→$s3

Instruction	C	MIPS
And	`a = b & c;`	`and  $s1,$s2,$s3`
And Immediate	`a = b & 0x1;`	`andi $s1,$s2,0x1`
Or	`a = b | c;`	`or   $s1,$s2,$s3`
Or Immediate	`a = b | 0x5;`	`ori  $s1,$s2,0x5`
Not Or	`a = ~(b | c);`	`nor  $s1,$s2,$s3`
Exclusive Or	`a = b ^ c;`	`xor  $s1,$s2,$s3`
Exclusive Or Immediate	`a = b ^ 0xF;`	`xori $s1,$s2,0xF`

Shift Instructions

- In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
  - Shift operations are faster
  - Does not work with shifting right/division
- *Logical shift*: Add zeros as you shift
- *Arithmetic shift*: Sign-extend as you shift
  - Only applies when you shift right (preserves sign)
- Can shift by immediate or value in a register

Shift Instructions

Instruction Name	MIPS
Shift Left Logical	`sll  $s1,$s2,1`
Shift Left Logical Variable	`sllv $s1,$s2,$s3`
Shift Right Logical	`srl  $s1,$s2,2`
Shift Right Logical Variable	`srlv $s1,$s2,$s3`
Shift Right Arithmetic	`sra  $s1,$s2,3`
Shift Right Arithmetic Variable	`srav $s1,$s2,$s3`

- When using immediate, only values 0-31 are accepted
- When using variable, only lowest 5 bits are used (read as unsigned)

Shift Instructions

```
# sample shift instructions
addi $t0,$0 ,-256 # $t0=0xFFFFFF00
sll  $s0,$t0,3    # $s0=0xFFFFF800
srl  $s1,$t0,8    # $s1=0x00FFFFFF
sra  $s2,$t0,8    # $s2=0xFFFFFFFF

addi $t1,$0 ,-22  # $t1=0xFFFFFFEA
                  # low 5: 0b01010
sllv $s3,$t0,$t1  # $s3=0xFFFC0000
# same as sll $s3,$t0,10
```

Shift Instructions

- Example 1:

```
# lb using lw:  lb $s1,1($s0)
lw   $s1,0($s0)  # get word
andi $s1,$s1,0xFF00 # get 2nd byte
srl  $s1,$s1,8    # shift into lowest
```

Shift Instructions

- Example 2:

```
# sb using sw:  sb $s1,3($s0)
lw   $t0,0($s0)  # get current word
andi $t0,$t0,0xFFFFFF # zero top byte
sll  $t1,$s1,24  # shift into highest
or   $t0,$t0,$t1 # combine
sw   $t0,0($s0)  # store back
```

Inequalities in MIPS

- Inequality tests: <, <=, >, and >=
  - RISC: implement all with 1 additional instruction
- Set on Less Than (`slt`)
  - **`slt dst,src1,src2`**
  - Stores 1 in **`dst`** if value in **`src1`** < value in **`src2`** and stores 0 in **`dst`** otherwise
- Combine with **`bne`**, **`beq`**, and **`$0`**

Inequalities in MIPS

- C Code:

```
if (a <b) {
  ... /* then */
}
```

(let a→$s0, b→$s1)

- MIPS Code:

```
slt $t0,$s0,$s1
# $t0=1 if a<b
# $t0=0 if a>=b
bne $t0, $0,then
# go to then
#   if $t0≠0
```

Inequalities in MIPS

- C Code:

```
if (a >=b) {
  ... /* then */
}
```

(let a→$s0, b→$s1)

- MIPS Code:

```
slt $t0,$s0,$s1
# $t0=1 if a<b
# $t0=0 if a>=b
beq $t0, $0,then
# go to then
#   if $t0=0
```

Immediates in Inequalities

- Three variants of `slt`:
  - `sltu  dst,src1,src2`: unsigned comparison
  - `slti  dst,src,imm`: compare against constant
  - `sltiu dst,src,imm`: unsigned comparison against constant
- Example:

```
addi  $s0,$0,-1   # $s0=0xFFFFFFFF
slti  $t0,$s0,1   # $t0=1
sltiu $t1,$s0,1   # $t1=0
```

Assembly Language vs. Machine Language

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

Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it’ll overwrite whatever the program has put into it
- Solution:
  - Reserve a register (**$1** or **$at** for “assembler temporary”) that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it’s not safe to code with it

---

MIPS: Software conventions for Registers

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	. . .		(callee must save)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont’d)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
. . .		(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

MIPS Branch Machine Instructions

- Compare and Branch
  - BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset<>
- Compare to zero and Branch
  - BLEZ rs, offsetif R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset>
  - BLTZ rs, offset<
  - BGEZ rs, offset>=
  - BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL rs, offset >=!
- Remaining set of compare and branch ops take two instructions
- Almost all comparisons are against zero!

Delayed Branches

```
      li     r3, #7
      sub    r4, r4, 1
      bz     r4, LL
      addi   r5, r3, 1      ⇐Delay Slot Instruction
      subi   r6, r6, 2
LL:   slt    r1, r3, r5
```

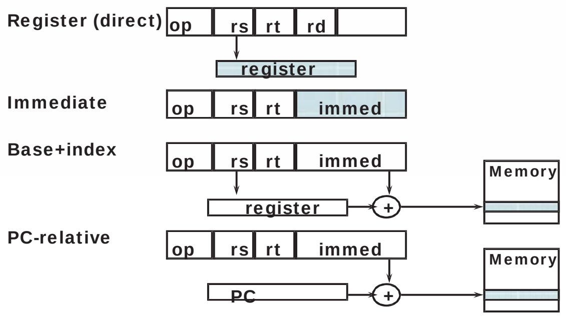
- In the **Raw** MIPS, the instruction after the branch is executed even when the branch is taken
  - This is hidden by the assembler for the MIPS **virtual machine**
  - allows the compiler to better utilize the instruction pipeline (???)
- Jump and link (jalinst):
  - Put the return addr. Into link register (R31):
    - PC+4 (logical architecture)
    - PC+8 physical (Raw) architecture ⇒delay slot executed
  - Then jump to destination address

Miscellaneous MIPS I Instructions

- **break**A breakpoint trap occurs, transfers control to exception handler
- **syscall**A system trap occurs, transfers control to exception handler
- **coprocessor instrs.**Support for floating point
- **TLB instructions**Support for virtual memory: discussed later
- **restore from exception** Restores previous interrupt mask & kernel/user mode bits into status register
- **load word left/right**Supports misaligned word loads
- **store word left/right**Supports misaligned word stores

MIPS Addressing Modes/Instruction Formats

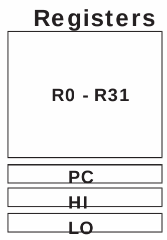
- All instructions 32 bits wide



- Register Indirect?

MIPS R3000 Instruction Set Architecture (Summary)

- Register Set
  - 32 general 32-bit registers
  - Register zero ($R0) always zero
  - Hi/Lo for multiplication/division
- Instruction Categories
  - Load/Store
  - Computational
    - Integer/Floating point
  - Jump and Branch
  - Memory Management
  - Special
- 3 Instruction Formats: all 32 bits wide



OP	rs	rt	rd	sa	funct

OP	rs	rt	immediate

OP	jump target

Summary: Salient features of MIPS I

- **32-bit fixed format inst**(3 formats)
- **32 32-bit GPR** (R0 contains zero) and **32 FP registers** (and **HI LO**)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr**.
- **Single address mode for load/store**: base+ displacement (offset)
  - no indirection, scaled
- **16-bit immediate** plus LUI
- **Simple branch conditions**
  - compare against zero or two registers for =,≠
  - no integer condition codes
- **Delayed branch**
  - execute instruction after a branch (or jump) even if the branch is taken

    (Compiler can fill a delayed branch with useful work about 50% of the time)

MIPS delayed branch too much implementation exposure at the ISA level

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Justin Hsia (UCB)