Instruction Set Architecture (ISA) and Assembly Language

Tyler Bletsch
Duke University

Slides are derived from work by Daniel J. Sorin (Duke), Alvy Lebeck (Duke), and Amir Roth (Penn)
Instruction Set Architecture (ISA)

- ISAs in General
  - Using MIPS as primary example
  - MIPS Assembly Programming
- Other ISAs
Outline

• What is an ISA?
• Assembly programming (in the MIPS ISA)
• Other ISAs
What Is a Computer?

• Machine that has storage (to hold instructions and data) and that executes instructions

• Storage (as seen by each running program)
  • Memory:
    • $2^{32}$ bytes for 32-bit machine
    • $2^{64}$ bytes for 64-bit machine  
      \[
      \text{[[ impossible! mystery for later... ]]}\]
  • Registers: a few dozen 32-bit (or 64-bit) storage elements
    • Live inside processor core

• Instructions
  • Move data from memory to register or from register to memory
  • Compute on values held in registers
  • Switch to instruction other than the next one in order
  • Etc.
What Is An ISA?

- **Functional & precise** specification of computer
  - What storage does it have? How many registers? How much memory?
  - What instructions does it have?
  - How do we specify operands for instructions?

- ISA = “**contract**” between software and hardware
  - Sort of like a “hardware API”
  - Specifies what hardware will do when executing each instruction

And how do we specify these in bits?
Architecture vs. Microarchitecture

- ISA specifies WHAT hardware does, not HOW it does it
  - No guarantees regarding these issues:
    - How operations are implemented
    - Which operations are fast and which are slow
    - Which operations take more power and which take less
  - These issues are determined by the **microarchitecture**
    - Microarchitecture = how hardware implements architecture
    - Can be any number of microarchitectures that implement the same architecture (Pentium and Core i7 are almost the same architecture, but are very different microarchitectures)

- Strictly speaking, ISA is the architecture, i.e., the interface between the hardware and the software
  - Less strictly speaking, when people talk about architecture, they’re also talking about how the architecture is implemented
Von Neumann Model of a Computer

- Implicit model of all modern ISAs
  - “von NOY-man” (German name)
  - Everything is in memory (and perhaps elsewhere)
    - instructions and data
- Key feature: **program counter (PC)**
  - PC is the memory address of the currently executing instruction
  - Next PC is \( PC + \text{length_of_instruction} \) unless instruction specifies otherwise
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts
An Abstract 32-bit Von Neumann Architecture

- Fetch instruction from PC
- Decode instruction
- Execute instruction
  - Read input operand(s)
    - (registers and/or memory locations and/or “immediates”)
  - Perform operation on input operands
  - Write result, if any, in output operand
    - (register or memory location)
  - Change PC to next instruction

- (32-bit) PC
- registers (each register holds one 32-bit operand)
- Memory
  - $2^{32}$ bytes
  - Holds instructions and data

32-bit address of current instruction
Outline

- What is an ISA?
- Assembly programming (in the MIPS ISA)
- Other ISAs
OK, so what does this assembly code mean? Let’s dig into each line ...
Simple, Running Example

loop: lw $1, Memory[1004]
lw $2, Memory[1008]
add $3, $1, $2
add $4, $4, $3
j loop

NOTES
“loop:” = line label (in case we need to refer to this instruction’s PC)
lw = “load word” = read a word (32 bits) from memory
$1 = “register 1” → put result read from memory into register 1
Memory[1004] = address in memory to read from (where x lives)

Note: almost all MIPS instructions put destination (where result gets written) first (in this case, $1)
Simple, Running Example

loop:    lw  $1, Memory[1004]
         lw  $2, Memory[1008]
         add  $3, $1, $2
         add  $4, $4, $3
         j  loop

NOTES
lw = “load word” = read a word (32 bits) from memory
$2 = “register 2” → put result read from memory into register 2
Memory[1008] = address in memory to read from (where y lives)
Simple, Running Example

```asm
loop:  lw  $1, Memory[1004]
      lw  $2, Memory[1008]
      add $3, $1, $2
      add $4, $4, $3
      j  loop
```

**NOTES**

add $3, $1, $2 = add what’s in $1 to what’s in $2 and put result in $3
Simple, Running Example

loop:  lw $1, Memory[1004]
lw $2, Memory[1008]
add $3, $1, $2
add $4, $4, $3
j loop

NOTES
add $4, $4, $3 = add what’s in $4 to what’s in $3 and put result in $4

Note: this instruction overwrites previous value in $4
Simple, Running Example

```assembly
loop:   lw $1, Memory[1004]
       lw $2, Memory[1008]
       add $3, $1, $2
       add $4, $4, $3
       j loop
```

**NOTES**

- `j` = “jump”
- `loop` = PC of instruction at label “loop” (the first lw instruction above) sets next PC to the address labeled by “loop”

Note: all other instructions in this code set next PC = PC+1
Assembly Code Format

• Every line of program has:
  - label (optional) – followed by “:”
  - instruction
  - comment (optional) – follows “#”

```assembly
loop:    lw   $1, Memory[1004]    # read from address 1004
         lw   $2, Memory[1008]
         add  $3, $1, $2
         add  $4, $4, $3
         j    loop                  # jump back to instruction at label loop
```

Note: a label is just a convenient way to represent an address so programmers don’t have to worry about numerical addresses.

Also, you don’t indent instructions to “nest” them – it’s flat list.
Assembly $\leftrightarrow$ Machine Code

- Every MIPS assembly instruction has a unique 32-bit representation
  - `add $3, $2, $7` $\leftrightarrow$ 00000000010011110001100000100000
  - `lw $8, Mem[1004]` $\leftrightarrow$ 10001100000010000000001111001100

- Computer hardware deals with bits
- We find it easier to look at the assembly
  - But they’re equivalent! No magical transformation.

- So how do we represent each MIPS assembly instruction with a string of 32 bits?
MIPS Instruction Format

| opcode (6 bits) | operands (26 bits) |

- **opcode** = what type of operation to perform
  - add, subtract, load, store, jump, etc.
  - 6 bits → how many types of operations can we specify?
- **operands** specify: inputs, output (optional), and next PC (optional)
- **operands** can be specified with:
  - register numbers
  - memory addresses
  - **immediates** (values wedged into last 26 bits of instruction)
### MIPS Instruction Formats

- 3 variations on theme from previous slide
  - All MIPS instructions are either R, I, or J type
  - Note: all instructions have opcode as first 6 bits

<table>
<thead>
<tr>
<th>R-type</th>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Rd(5)</th>
<th>Sh(5)</th>
<th>Func(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>Op(6)</td>
<td>Rs(5)</td>
<td>Rt(5)</td>
<td>Immed(16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td>Op(6)</td>
<td>Target(26)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### MIPS Format – R-Type Example

- **add $1, $2, $3**  
  # $1 = $2 + $3
- **add Rd, Rs, Rt**  
  # d=dest, s=source, t=??
- **Op = 6-bit code for “add” = 000000**
- **Rs = 00010**
- **Rt = 00011**
- **Rd = 00001**
- **don’t worry about Sh and Func fields for now**

<table>
<thead>
<tr>
<th>R-type</th>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Rd(5)</th>
<th>Sh(5)</th>
<th>Func(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>000000</td>
<td>00010</td>
<td>00011</td>
<td>00001</td>
<td></td>
<td>000010000</td>
</tr>
</tbody>
</table>

**Note:** the MIPS architecture has 32 registers. Therefore, it takes \( \log_2{32} = 5 \) bits to specify any one of them.

If you’re looking back at this slide later on: Okay, let’s talk about Sh and Func.  
Sh is just the shift amount, used only for bit shifting instructions (sll, srl, sra).  
Func is the interesting one. In order to allow more opcodes than just the \( 2^6 = 64 \) that you’d expect, opcode 000000 is special. It means “R-type instruction whose actual verb is given in the Func field”. So an R-type add is encoded with Op=000000 and Func=100000.  
Here’s two references with the actual tables involved: [this page covers it well](#), [this book excerpt shows it in Fig A.10.2](#).
Uh-Oh

- Let’s try a lw (load word) instruction
- lw $1, Memory[1004]
  - 6 bits for opcode
  - That leaves 26 bits for address in memory
- But an address is 32 bits long!
  - What gives?
Memory Operand Addressing (for loads/stores)

- We have to use indirection to specify memory operands
- **Addressing mode**: way of specifying address
  - (Register) Indirect:  `lw $1,($2)`  # $1=memory[$2]
  - Displacement:  `lw $1,8($2)`  # $1=memory[$2+8]
  - Index-base:  `lw $1,($2,$3)`  # $1=memory[$2+$3]
  - Memory-indirect:  `lw $1,@($2)`  # $1=memory[memory[$2]]
  - Auto-increment:  `lw $1,($2)+`  # $1=memory[$2++]

  ^ Last three not supported in MIPS

- What high-level language idioms are these used for?
MIPS Addressing Modes

- MIPS implements only displacement addressing mode
  - Why? Experiment on VAX (ISA with every mode) found distribution
    - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
    - 80% use displacement or register indirect (=displacement 0)

- I-type instructions: 16-bit displacement
  - Is 16-bits enough?
  - Yes! VAX experiment showed 1% accesses use displacement >2^{15}

```
I-type

<table>
<thead>
<tr>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Immed(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>00010</td>
<td>00001</td>
<td>000000000000001000</td>
</tr>
</tbody>
</table>

lw $1, 8($2)
```
Back to the Simple, Running Example

- assume $6=1004=address of variable x in C code example
- and recall that 1008=address of variable y in C code example

```
loop:  lw $1, Memory[1004]  ⇒  lw $1, 0($6)  # put val of x in $1
       lw $2, Memory[1008]  ⇒  lw $2, 4($6)  # put val of y in $2
       add $3, $1, $2
       add $4, $4, $3
       j loop
```
MIPS Format – I-Type Example

- **I-type**
  
<table>
<thead>
<tr>
<th>I-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op(6)</td>
</tr>
<tr>
<td>Rs(5)</td>
</tr>
<tr>
<td>Rt(5)</td>
</tr>
<tr>
<td>Immed(16)</td>
</tr>
</tbody>
</table>

- `lw $1, 0($6) // $1 = Memory [\$6 + 0]`
  - `lw Rt, immed(Rs)`
  - Opcode = 6-bit code for “load word” = 100011
  - Rs = 6 = 00110
  - Rt = 1 = 00001
  - Immed = 0000 0000 0000 0000 = 0_{10}

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>immed</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>00110</td>
<td>00001</td>
<td>000000000000000000000000000000000000000000</td>
</tr>
</tbody>
</table>
**Memory Addressing Issue: Alignment**

- **Alignment**: require that objects fall on address that is multiple of their size
- **32-bit integer**
  - Aligned if address % 4 = 0 [ % is symbol for “mod” ]
    - (Binary ends in 00)
    - (Hex ends in 0, 4, 8, or C)
- **64-bit integer?**
  - Aligned if ?
- **Question**: what to do with unaligned accesses (uncommon case)?
  - Support in hardware? Makes all accesses slower
  - Trap to software routine? Possibility
  - **MIPS? ISA support**: unaligned access using two instructions:
    \[ \text{ulw} @XXXX10 = \text{lwl} @XXXX10; \text{lwr} @XXXX10 \]
Declaring Space in Memory for Data

- Add two numbers x and y:

```assembly
.text
main:
    la $3, x
    lw $4, 0($3)
    la $3, y
    lw $5, 0($3)
    add $6, $4,$5

.data
x: .word 10
y: .word 3
emptystr: .space 32
hellostr: .asciiz "hello"
```

What memory region?
- Static (AKA global)
MIPS Operand Model

- MIPS is a “load-store” architecture
  - All computations done on values in registers
    - Can only access memory with load/store instructions
  - 32 32-bit integer registers
    - Actually 31: $0$ is hardwired to value 0
    - Also, certain registers conventionally used for special purposes
      - We’ll talk more about these conventions later
  - 32 32-bit Floating Point registers
    - Can also be treated as 16 64-bit Floating Point registers
  - HI,LO: destination registers for multiply/divide
How Many Registers?

- Registers faster than memory → have as many as possible? No!
  - One reason registers are faster is that there are fewer of them
    - Smaller storage structures are faster (hardware truism)
  - Another is that they are directly addressed (no address calc)
    - More registers → larger specifiers → fewer regs per instruction
- Not everything can be put in registers
  - Structures, arrays, anything pointed-to
    - Although compilers are getting better at putting more things in
  - More registers means more saving/restoring them
    - At procedure calls and context switches
- Number of registers:
  - 32-bit x86: 8
  - MIPS32: 32
  - ARM: 16
  - 64-bit x86: 16 (plus some weird special purpose ones)
MIPS Register Usage/Naming Conventions

- It turns out the registers also have **names** as well as numbers.
- Why? We’ll see later.
  - For now: $s$ and $t$ are general purpose, others are special-purpose.

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
</tr>
<tr>
<td>2</td>
<td>v0</td>
</tr>
<tr>
<td>3</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>a0</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0</td>
</tr>
<tr>
<td>16</td>
<td>s0</td>
</tr>
<tr>
<td>17</td>
<td>. . .</td>
</tr>
<tr>
<td>18</td>
<td>s1</td>
</tr>
<tr>
<td>19</td>
<td>. . .</td>
</tr>
<tr>
<td>20</td>
<td>s3</td>
</tr>
<tr>
<td>21</td>
<td>. . .</td>
</tr>
<tr>
<td>22</td>
<td>s5</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp</td>
</tr>
<tr>
<td>29</td>
<td>sp</td>
</tr>
<tr>
<td>30</td>
<td>fp</td>
</tr>
<tr>
<td>31</td>
<td>ra</td>
</tr>
</tbody>
</table>

- zero constant
- at reserved for assembler
- v0 expression evaluation & function results
- a0 arguments
- k0 reserved for OS kernel
- gp pointer to global area
- sp stack pointer
- fp frame pointer
- ra return address
System calls

• How do we do Input/Output (read/print from console, etc.)?
• We ask an **Operating System** (OS) to do it for us
  • The OS manages hardware so we don’t have to worry about it
• How do we, a user program, ask the OS for something?
  • Special hardware support: **system calls**

• It’s like a summoning ritual:
  1. Place the right values in certain registers
  2. Believe
  3. Issue a `syscall` instruction
  4. Thing you wanted to print gets printed, thing you wanted to read appears in a register, etc.
System Call Instruction

• What does our system call ritual look like?
  1. Put a “system call code” into register $v0
     • Example: if $v0==1, then syscall will print an integer
  2. Put arguments (if any) into registers $a0, $a1, or $f12 (for floating point)
  3. Issue the syscall
  4. Results returned in registers $v0 or $f0 (for floating point)
### SPIM System Call Support

<table>
<thead>
<tr>
<th>code</th>
<th>service</th>
<th>ArgType</th>
<th>Arg/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>print</td>
<td>int</td>
<td>Provide int in $a0</td>
</tr>
<tr>
<td>2</td>
<td>print</td>
<td>float</td>
<td>Provide float in $f12</td>
</tr>
<tr>
<td>3</td>
<td>print</td>
<td>double</td>
<td>Provide double in $f12 &amp; $f13</td>
</tr>
<tr>
<td>4</td>
<td>print</td>
<td>string</td>
<td>Provide string buffer address $a0</td>
</tr>
<tr>
<td>5</td>
<td>read</td>
<td>integer</td>
<td>Get back int in $v0</td>
</tr>
<tr>
<td>6</td>
<td>read</td>
<td>float</td>
<td>Get back float in $f0</td>
</tr>
<tr>
<td>7</td>
<td>read</td>
<td>double</td>
<td>Get back double in $f0 &amp; $f1</td>
</tr>
<tr>
<td>8</td>
<td>read</td>
<td>string</td>
<td>Provide $a0=buffer, $a1=length</td>
</tr>
<tr>
<td>9</td>
<td>sbrk</td>
<td>$a0=amount</td>
<td>Get back address in $v0</td>
</tr>
<tr>
<td>10</td>
<td>exit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Asks the OS to allocate some memory on the heap. This is what malloc is based on. In fact, as far as you’re concerned, **in MIPS, the sbrk syscall is your malloc!**

Plus a few more for general file IO which we shouldn’t need.
Syscall example

# addone.s: Take in an integer, then return that integer plus one.
# Example for ECE/CS 250
# Updated by Tyler Bletsch 2022-05-18

.text
.align 2
.globl main
main:
    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    # the number is now in $v0

    addi $t0, $v0, 1
    # Add one to the number and put the result in $t0 for use later

    li $v0, 4
    la $a0, response
    syscall
    # > Print a prefix message for the output

    move $a0, $t0
    # Copy the number to be printed from $t0 into $a0,
    # since that's where it has to be for the syscall

    li $v0, 1
    syscall
    # > Print the number in $a0

    li $v0, 0
    jr $ra
    # \ return 0 from main

.end main
#  end of main function

.data
# Start of data segment

prompt: .asciiz "Give me a number: "
response: .asciiz "That number plus one is: "

Control Instructions – Changing the PC

• Most instructions set next PC = PC+1
• But what about handling control flow?
• Conditional control flow: if condition is satisfied, then change control flow
  • if/then/else
  • while() loops
  • for() loops
  • switch
• Unconditional control flow: always change control flow
  • procedure calls

• How do we implement control flow in assembly?
Control Instructions

• Three issues:
  1. Testing for condition: Is PC getting changed?
  2. Computing target: If so, then where to?
  3. Dealing with procedure calls (later)

• Types of control instructions
  • conditional branch: `beq`, `beqz`, `bgt`, etc.
    • if condition is met, “branch” to some new PC; else PC=PC+1
    • many flavors of branch based on condition (<, >0, <=, etc.)
  • unconditional jump: `j`, `jr`, `jal`, `jalr`
    • change PC to some new PC
    • several flavors of jump based on how new PC is specified
Control Instructions I: Condition Testing

- Three options for **testing conditions**
  - **Option I:** *implicit condition codes (CCs)* (not used in MIPS except for floats)
    
    ```
    subi $2, $1, 10   // sets “negative” CC
    bn target   // if negative CC set, goto target
    # bn = “Branch if Negative”
    ```
  
  - **Option II:** *compare and branch instructions* (sorta used in MIPS)
    
    ```
    beq $1, $2, target   // if $1==$2, goto target
    # beq = “Branch if Equal”
    ```
  
  - **Option III:** *condition registers, separate branch insns* (in MIPS)
    
    ```
    slti $2, $1, 10   // set $2 if $1<10
    # slti = “Set Less-Than Immediate”
    bnez $2, target   // if $2 ! = 0, goto target
    # bnez = “Branch if Not-Equal to Zero”
    ```
MIPS Conditional Branches

- MIPS uses combination of options II and III
  - (II) Compare 2 registers and branch: `beq`, `bne`
    - Equality and inequality only
      + Don’t need adder for comparison
  - (II) Compare 1 register to zero and branch: `bgtz`, `bgez`, `bltz`, `blez`
    - Greater/less than comparisons
      + Don’t need adder for comparison
  - (III) Set explicit condition registers: `slt`, `sltu`, `slti`, `sltiu`, etc.

- Why?
  - 86% of branches in programs are (in)equalities or comparisons to 0
  - OK to take two insns to do remaining 14% of branches
    - Make the common case fast (MCCF)!
Control Instructions II: Computing Target

- Three options for **computing targets** (target = next PC)
  - Option A: **PC-relative** (next PC = current PC +/- some value)
    - Position-independent within procedure
    - Used for branches and jumps within a procedure
  - Option B: **Absolute** (next PC = some value)
    - Position independent outside procedure
    - Used for procedure calls
  - Option C: **Indirect** (next PC = contents of a register)
    - Needed for jumping to dynamic targets
    - Used for returns, dynamic procedure calls, switches

- How far do you need to jump?
  - Typically not so far within a procedure (they don’t get very big)
  - Further from one procedure to another
MIPS: Computing Targets

- MIPS uses all 3 ways to specify target of control insn
  - PC-relative → conditional branches: `bne, beq, blez`, etc.
    - 16-bit relative offset, <0.1% branches need more
    - \( PC = PC + 4 + \text{immediate} \) if condition is true (else \( PC = PC + 4 \))

- Absolute → unconditional jumps: `j target`
  - 26-bit offset (can address \( 2^{28} \) words < \( 2^{32} \) → what gives?)

- Indirect → Indirect jumps: `jr $31`
• Control idiom: **if-then-else**

```plaintext
if (A < B) A++;    // assume A in register $1
else B++;         // assume B in $2

slt $3,$1,$2      // if $1<$2, then $3=1
beqz $3,else      // branch to else if !condition
addi $1,$1,1
j endif           // jump to endif

else:
    addi $2,$2,1
endif:
```

General form of if/then/else
1. If <CONDITION> is false, branch to else
2. <THEN-BODY>
3. j endif
4. else:
5. <ELSE-BODY>
6. endif:

General form of if/then (no else)
1. If <CONDITION> is false, branch to endif
2. <THEN-BODY>
3. endif:
Control Idiom: While loop

- Control idiom: **while loop**

  ```
  while (A < B) { A++; }  // assume A in register $1
  ```

  **loop:**
  ```
  slt $3,$1,$2       // if $1<$2, then $3=1
  beqz $3,endloop  // branch to escape if !condition
  addi $1,$1,1
  j  loop          // loop to top
  ```

  **endloop:**

---

**General form of 'while' loop**
1. loop:
2. If <CONDITION> is false, branch to endloop
3. <BODY>
4. j loop
5. endloop:
Control Idiom: Arithmetic For Loop

- Second idiom: "for loop" with arithmetic induction

```c
int A[100], sum, i, N;
for (i=0; i<N; i++) {
    // assume: i in $1, N in $2
    sum += A[i];  // &A[i] in $3, sum in $4
}
```

```assembly
li $1, 0  # initialize i to 0
# Not shown: initialize $3 to A (address of start of array)
loop:    slt $8,$1,$2  # if i<N, then $8=1; else $8=0
         beqz $8,endloop  # test for exit at loop header
         lw $9,0($3)  # $9 = A[i] (not &A[i])
         add $4,$4,$9  # sum = sum + A[i]
         addi $3,$3,4  # increment &A[i] by sizeof(int)
         addi $1,$1,1  # i++
         j loop  # backward jump
endloop:
```

General form of ‘for’ loop:
1. Do <INITIALIZER>
2. loop:
3. If <CONDITION> is false, branch to endloop
4. <BODY>
5. <INCREMENTER>
6. j loop
7. endloop:
Revisiting Control Idioms: Pointer For Loop

- **Control idiom:** *for loop with pointer induction*

```
struct node_t { int val; struct node_t *next; };
struct node_t *p, *head;
int sum=0;
for (p=head; p!=NULL; p=p->next) // p in $1, head in $2
    sum += p->val // sum in $3

li $3, 0 # sum = 0
move $1,$2 # p = head
loop: beq $1,$0,endloop # if p==0 (NULL), goto exit
lw $5,0($1) # $5 = *p = p→val
add $3,$3,$5 # sum = sum + p→val
lw $1,4($1) # p = *(p+1) = p→next
j loop # go back to top of loop
endloop:
```

**General form of ‘for’ loop**

1. Do <INITIALIZER>
2. loop:
3. If <CONDITION> is false, branch to endloop
4. <BODY>
5. <INCREMENTER>
6. j loop
7. endloop:
Some of the Most Important Instructions

- **Math/logic**
  - add, sub, mul, div

- **Access memory**
  - \( \text{lw} = \text{load (read) word:} \)
    \[ \text{lw} \; $3, \; 4($5) \quad \# \quad $3 = \text{memory}[$5+4] \]
  - \( \text{sw} = \text{store (write) word:} \)
    \[ \text{sw} \; $3, \; 4($5) \quad \# \quad \text{memory}[$5+4] = $3 \]

- **Change PC, perhaps conditionally**
  - **Branches:** blt, bgt, beqz, etc.
  - **Jumps:** j, jr, jal (will see last two later)

- **Handy miscellaneous instructions**
  - \( \text{la} = \text{load address} \)
  - \( \text{move: move} \; $1, \; $5 \quad \# \; \text{copies (doesn’t move!) $5 into } $1 \)
  - \( \text{li} = \text{load immediate:} \)
    \[ \text{li} \; $5, \; 42 \quad \# \; \text{writes value 42 into } $5 \]
    (terrible name for instr!! not a load – no memory access!)

*Note: sw is unusual in that the destination of instruction isn’t first operand!*
Clarifying “load” instructions (1)

<table>
<thead>
<tr>
<th>C code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.data</td>
</tr>
<tr>
<td></td>
<td>array: .word 55, 27, 19, 88</td>
</tr>
<tr>
<td></td>
<td>str: .asciiz &quot;hello&quot;</td>
</tr>
<tr>
<td></td>
<td>.text</td>
</tr>
<tr>
<td>int main() {</td>
<td>main:</td>
</tr>
<tr>
<td>int r1 = 5;</td>
<td>li $1, 5</td>
</tr>
<tr>
<td>int* r2 = array;</td>
<td>la $2, array</td>
</tr>
<tr>
<td>int r3 = *r2;</td>
<td>lw $3, 0($2)</td>
</tr>
<tr>
<td>int r4 = r1;</td>
<td>move $4, $1</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

- “Load immediate” isn’t really a load (it doesn’t come from memory)
- “Load address” is just a “load immediate”, but the assembler figures out the immediate from labels
- “Move” just copies values between registers
- Of the instructions shown, only “load word” actually loads from memory
Clarifying “load” instructions (2)

Load immediate instruction: \texttt{li}

Load immediate instruction: \texttt{la}

Move instruction: \texttt{move}

Load word instruction: \texttt{lw}
What about store word (sw)?

Difference between lw and sw:
- lw goes from memory to register
- sw goes from register to memory
Flavors of Load/Store Instructions

- We already know about `lw` and `sw`
  - `lw $3, 12($5)`
  - `sw $4, -4($6)`
- Also have load/store instructions that operate at non-word-size granularity
  - `lb = load byte, lh = load halfword`
  - `sb = store byte, sh = store halfword`
- Loads can access smaller size but always write all 32 bits of destination register
  - By default, sign-extend to fill register
  - Unless specified as unsigned with instrs: `lbu, lhu`
• Consider this struct:

```c
struct student {
    int id;  // size = 4
    char name[16];  // size = 16
    float gpa;  // size = 4
}
```

If we have a pointer to this struct in $1$, then:

- Can load the id with: `lw $2, 0($1)`
- Can load the first character of the name with: `lb $2, 4($1)`
  
  Why just first character? It's a string! Can't ever load it all into a register!
- Can load the float with: `l.s $f2, 20($1)`
  
  What's this `l.s` thing? It's like `lw`, but for floats. You'll learn float instructions on your own in Homework 2.
Many Other Operations

- Many types of operations
  - Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  - FP arithmetic: add, sub, mul, div, sqrt
  - Integer logical: and, or, xor, not, sll, srl, sra
  - Packed integer: padd, pmul, pand, por... (saturating/wraparound)

- What other operations might be useful?
- More operation types == better ISA??
- DEC VAX computer had LOTS of operation types
  - E.g., instruction for polynomial evaluation (no joke!)
  - But many of them were rarely/never used (ICQ: Why not?)
  - We’ll talk more about this issue later ...
Flavors of Math Instructions

• We already know about add
  • add $3, $4, $5

• Also have addi = “add immediate” [Note: I-type instr]
  • addi $3, $4, 42  #  $3 = $4 + 42

• And addu = “add unsigned”
  • addu $3, $4, $5
    #  same as add, but treat values as unsigned ints

• And even addiu = “add immediate unsigned”
  • addiu $3, $4, 42

• Same variants for sub, etc.
Datatypes

- Datatypes
  - Software view: property of data
  - Hardware view: data is just bits, property of operations
    - Same 32 bits could be interpreted as int or as instruction, etc.

- Hardware datatypes
  - Integer: 8 bits (byte), 16b (half), 32b (word), 64b (long)
  - IEEE754 FP: 32b (single-precision), 64b (double-precision)
  - Packed integer: treat 64b int as 8 8b int’s or 4 16b int’s
  - Packed FP
Procedure Calls: A Simple, Running Example

main:   li $1, 1  # $1 = 1
        li $2, 2  # $2 = 2
        $3 = call foo($1, $2)  # this is NOT actual MIPS code
        add $4, $3, $3
        {rest of main}
        {end program}

foo:    sub $5, $1, $2
        return ($5)

--------------------------------------------------
main is the caller
foo is the callee
Procedure Calls: Jump-and-Link and Return

main:  
li $1, 1  
li $2, 2  
$3 = call foo($1, $2)  
jal foo  
# jal = jump and link  
add $4, $3, $3  
{rest of main}

foo:  
sub $5, $1, $2  
return ($5)  
jr $ra

-----------------------------
jal does two things:
  1) sets PC = foo (just like a regular jump instruction)
  2) “links” to PC after the jal  
MIPS designates $31 for a special purpose: it’s the return address ($ra)

jr sets PC to the value in $ra  
computer executes add instr after jal
Procedure Calls: Why Link?

main:
li $1, 1
li $2, 2
$3 = call foo($1, $2) \rightarrow j foo \quad \# j = \text{jump}

r1:
add $4, $3, $3
add $1, $1, $4
j foo

r2:
sub $2, $1, $3
{rest of main}

foo:
sub $5, $1, $2
return ($5) \rightarrow \text{OK, now what?? Jump to r1? Jump to r2?}

Since function can be called from multiple places, must explicitly remember (link!) where called from.
Procedure Calls: Passing Args & Return Values

main:
li $1, 1
li $2, 2
move $a0, $1    # pass first arg in $a0
move $a1, $2    # pass second arg in $a1
jal foo
add $4, $3, $3  → add $4, $v0, $v0   # return value in $v0 now
{rest of main}

foo:
sub $5, $a0, $a1
move $v0, $5     # pass return value in $v0
jr $ra

-----------------------------------------------
Must use specific registers for passing arguments and return values.
MIPS denotes $a0-$a3 as argument registers.
MIPS denotes $v0-$v1 as return value registers.
Passing Arguments by Value or by Reference

• Passing arguments
  • **By value**: pass contents \([3+4]\) in \(a0\)
    
    ```
    int n; // n in 4($3)
    foo(n);
    lw $a0,4($3)
    jal foo
    ```
  
  • **By reference**: pass address \(3+4\) in \(a0\)
    
    ```
    int n; // n in 4($3)
    bar(&n);
    addi $a0,$3,4
    jal bar
    ```
Procedures Must Play Nicely Together

main:
li $1, 1
li $2, 2
move $a0, $1
move $a1, $2
jal foo
add $4, $v0, $v0
add $6, $4, $1  # $1 should still be 1
{rest of main}

foo:
sub $5, $a0, $a1
li $1, 3  # $1 now equals 3
add $5, $5, $1
move $v0, $5
jr $ra

What would happen if main uses $1 after calling foo but foo also uses $1?
Not good, right? Let's see why ...
int main (){ 
    int x=1;  
    int y=2;  
    int z = foo(x,y); 
    z = z + x; 
}

int foo(int a1, int a2){ 
    // code written by other person
    return a1+a2; 
}
Procedures Must Play Nicely Together

main:
li $1, 1
li $2, 2
move $a0, $1
move $a1, $2
jal foo
add $4, $v0, $v0
add $6, $4, $1  # $1 should still be 1
{rest of main

foo:
sub $5, $a0, $a1
li $1, 3  # $1 now equals 3
add $5, $5, $1
move $v0, $5
jr $ra

This seems contrived. Why can’t the programmer of foo just not use $1 Problem solved, right?

Nope! In real-world, one person doesn’t write all of the software. My code must play well with your code.
Procedures Use the Stack

- In general, procedure calls obey **stack discipline**
  - Local procedure state contained in **stack frame**
  - Where we can save registers to avoid problem in last slide
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses

- Procedure stack is **in memory**
  - Starts at “top” of memory and grows down
Preserving Registers Across Procedures

main:
li $1, 1
li $2, 2
move $a0, $1
move $a1, $2
jal foo
add $4, $v0, $v0
add $6, $4, $1
{rest of main}

foo:
sub $5, $a0, $a1
li $1, 3
add $5, $5, $1
move $v0, $5
jr $ra

Stack pointer is address of bottom of current stack frame. Always held in register $sp.
Preserving Registers Across Procedures

**main:**
li $1, 1
li $2, 2
move $a0, $1
move $a1, $2
jal foo
add $4, $v0, $v0
add $6, $4, $1
{rest of main}

**foo:**
make frame (move stack ptr)
save $1 in stack frame
sub $5, $a0, $a1
li $1, 3
add $5, $5, $1
move $v0, $5
restore $1 from stack frame
destroy frame
jr $ra
Preserving Registers Across Procedures

main:
li $1, 1
li $2, 2
move $a0, $1
move $a1, $2
jal foo
add $4, $v0, $v0
add $6, $4, $1
{rest of main}

foo:
make frame  \rightarrow \ subi $sp, $sp, 4
save $1 on stack frame  \rightarrow \ sw $1, 0($sp)
sub $5, $a0, $a1
li $1, 3
add $5, $5, $1
move $v0, $5
restore $1 from stack frame  \rightarrow \ lw $1, 0($sp)
destroy frame  \rightarrow \ addi $sp, $sp, 4
jr $ra
Who Saves/Restores Registers?

main:
li $1, 1
li $2, 2
move $a0, $1
move $a1, $2
jal foo
add $4, $v0, $v0
add $6, $4, $1
{rest of main}

foo:
subi $sp, $sp, 4
sw $1, 0($sp)
sub $5, $a0, $a1
li $1, 3
add $5, $5, $1
move $v0, $5
lw $1, 0($sp)
addi $sp, $sp, 4
jr $ra

In this example, the callee (foo) saved/restored registers. But why didn’t the caller (main) do that instead?
Two strategies for ensuring good happy registers

**Caller-saved registers**
- Right before a call, save the registers whose values you care about to the stack
- Right after it returns, restore them

I don’t trust that function I’m calling not to screw up my registers…he might consider them to be temporary! I’ll back them up to the stack first and restore them after…

Wooo, the caller was right, I’m trashing these registers!!!!!!

**Callee-saved registers**
- At the top of the function, save registers you will modify to the stack
- Right before you return, restore them

I can trust the callee, he wouldn’t mess with my registers. They’re safe.

Oh whoops, I just used some of these registers! Luckily I saved them at the start…I’ll restore them right before I return and the caller will never know!

```assembly
func1:
    addiu $sp, $sp, -4
    ... sw $t0, 0($sp)     # t registers
    jal func2
    lw $t0, 0($sp)        # s registers
    ... addiu $sp, $sp, 4

func3:
    ... jal func4
    ...

func4:
    addiu $sp, $sp, -4
    sw $s0, 0($sp)         # s registers
    ... # use $s0 somehow
    lw $s0, 0($sp)
    addiu $sp, $sp, 4
```

68
### MIPS Register Usage/Naming Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
</tr>
<tr>
<td>2</td>
<td>v0</td>
</tr>
<tr>
<td>3</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>a0</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0</td>
</tr>
<tr>
<td>9</td>
<td>t1</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0</td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp</td>
</tr>
<tr>
<td>29</td>
<td>sp</td>
</tr>
<tr>
<td>30</td>
<td>fp</td>
</tr>
<tr>
<td>31</td>
<td>ra</td>
</tr>
</tbody>
</table>

Also 32 floating-point registers: $f0 .. $f31

Important: The only general purpose registers are the $s and $t registers.

Everything else has a specific usage:
$s_a = arguments, $v = return values, $ra = return address, etc.
MIPS/GCC Procedure Calling Conventions

**Calling Procedure**

- **Step-1: Pass the arguments**
  - First four arguments (arg0-arg3) are passed in registers $a0-$a3
  - Remaining arguments are pushed onto the stack (in reverse order, arg5 is at the top of the stack)

- **Step-2: Save caller-saved registers**
  - Save registers $t0-$t9 if they contain live values at the call site

- **Step-3: Execute a jal instruction**

- **Step-4: Restore any $t registers you saved**
Called Routine

- Step-1: Establish stack frame
  - Subtract the frame size from the stack pointer
    \[ \text{addiu}\ sp, sp, -<\text{frame\_size}> \]

- Step-2: Save callee-saved registers in the frame
  - Register $ra is saved if routine makes a call
  - Registers $s0-$s7 are saved if they are used
On return from a call

- Step-1: Put returned values in registers $v0 and $v1 (if values are returned)
- Step-2: Restore callee-saved registers
  - $ra, $s0 - $s7
- Step-3: Pop the stack
  - Add the frame size to $sp
    addiu $sp, $sp, <frame-size>
- Step-4: Return
  - Jump to the address in $ra
    jr $ra
Which flavor of register to use?

- When to use **callee-saved $s register** vs **caller-saved $t register**?
- Choose to minimize saving/restoring needed
  - Can get complicated in practice
- Simple heuristic (gives decent efficiency):
  - If your function calls another function, use $s registers
    (if you make 5 calls, you’d need to save/restore a $t register 5 times, this way you just save it once)
  - If your function does not call other functions, use $t registers
    (no need to save/restore at all!)

**Note:** $ra is considered a callee-saved register (like $s), and is trashed if your function makes a call! **Gotta save/restore $ra if you call anything!**
The importance of saving $ra

\[ f: \]

\[ \ldots \]

\[ \ldots \]

\[ \text{jal } g \]

\[ \ldots \]

\[ \ldots \]

\[ \text{jr } $ra \]

\[ g: \]

\[ \ldots \]

\[ \ldots \]

\[ \text{jal } h \]

\[ \ldots \]

\[ \ldots \]

\[ \text{jr } $ra \]

\[ h: \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \text{jr } $ra \]
The importance of saving $ra

f:  
   ...
   ...
   ...
   jal g
   ...
   ...
   jr $ra

g:  
   ...
   ...
   ...
   jal h
   ...
   ...
   jr $ra

h:  
   ...
   ...
   ...
   ...
   ...
   ...
   jr $ra
The importance of saving $ra

f:       

PC       g:       h:

...     ...     ...

...     ...     ...

jal g   jal h   ...

...     ...     ...

jr $ra   jr $ra   jr $ra
The importance of saving "$ra"

f:

...  
...  
jal g  
$ra  
...  
...  
jr $ra

h:

...  
...  
jal h  
PC  
...  
...  
jr $ra

PC:

...  
...  
jal h  
PC  
...  
...  
jr $ra
The importance of saving $ra

<table>
<thead>
<tr>
<th>f:</th>
<th>g:</th>
<th>h:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>jal g</td>
<td>jal h</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>$ra</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>jr $ra</td>
<td>jr $ra</td>
<td>jr $ra</td>
</tr>
</tbody>
</table>
The importance of saving $ra

f:

... ... ...
jal g ... ...
...
...
...
jr $ra


g:

... ...
jal h ...
...
...
...
...
...
jr $ra


h:

... ...
...
...
...
...
...
PC jr $ra
The importance of saving $ra

f:

... ...
...
jal g ...
...
jr $ra ...

PC $ra


g:

... ...
...
jal h ...
...
jr $ra ...


h:

... ...
...
...
...
jr $ra ...


jr $ra
The importance of saving $ra

<table>
<thead>
<tr>
<th>f:</th>
<th>g:</th>
<th>h:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>jal g</td>
<td>jal h</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>jr $ra</td>
<td>PC jr $ra</td>
<td>jr $ra</td>
</tr>
</tbody>
</table>
The importance of saving $ra

Where we should have gone!!

ERROR!

Because we didn’t save/restore $ra, we’re trapped in g forever!!
The importance of saving $ra

Let’s try this again, but we’ll save/restore $ra on the stack properly.
The importance of saving $ra

f:
<save $ra>
...
...
jal g
...
...
<restore $ra>
jr $ra

g:
<save $ra>
...
...
jal h
...
...
<restore $ra>
jr $ra

h:
...
...
jal h
...
...
<restore $ra>
jr $ra

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!
The importance of saving $ra

f:
  $ra
  <save $ra>
  ...
  ...
  jal g
  ...
  ...
  <restore $ra>
  jr $ra

PC: g:
  <save $ra>
  ...
  ...
  jal h
  ...
  ...
  <restore $ra>
  jr $ra

h:
  ...
  ...
  ...
  ...

Don’t need to save $ra in h because this function doesn't jal, so it doesn’t modify $ra!
The importance of saving $ra

f:
<save $ra>
...
jal g
...
<restore $ra>
jr $ra

The copy of $ra saved in g's stack

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!

h:
<save $ra>
...
jal h
...
<restore $ra>
jr $ra
The importance of saving $ra

f:
<save $ra>
...
...
jal g
...
<restore $ra>
jr $ra

g:
<save $ra>
...
...
jal h
...
<restore $ra>
jr $ra

h:
...
...
jal h
...
...
jr $ra

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!
The importance of saving $ra

f:
<save $ra>
...
...
jal g
...
<restore $ra>
jr $ra

g:
<save $ra>
...
...
jal h
...
<restore $ra>
jr $ra

h:

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!

The copy of $ra saved in g’s stack
The importance of saving $ra

f:
<save $ra>
...
...
jal g
...
...<restore $ra>
jr $ra

g:
<save $ra>
...
...
jal h
...
...<restore $ra>
jr $ra

h:
...
...
jal h
...
...<restore $ra>
jr $ra

PC jr $ra

The copy of $ra saved in g's stack

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!
The importance of saving $ra

f:
<save $ra>
...
...
jal g
...
<restore $ra>
jr $ra

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!

h:
<save $ra>
...
jal h
...
<restore $ra>
jr $ra

The copy of $ra saved in g’s stack
The importance of saving $ra

f:
<save $ra>
...
...
jal g
...
...
<restore $ra>
jr $ra

Don't need to save $ra in h because this function doesn't jal, so it doesn't modify $ra!

h:
<save $ra>
...
...
jal h
...
...
<restore $ra>
jr $ra

The copy of $ra saved in g's stack
The importance of saving $ra

f:

<save $ra>
...
...
jal g
...
<restore $ra>
jr $ra

g:

<save $ra>
...
...
jal h
...
<restore $ra>
jr $ra

h:

...
...
jal h
...
...
jr $ra

Don’t need to save $ra in h because this function doesn’t jal, so it doesn’t modify $ra!
The importance of saving $ra

f:

<save $ra>
...
...
jal g
...

<restore $ra>
jr $ra

PC $ra

Success!

Because we *did* save/restore $ra, we escaped g back to f!!
Functions vs inner labels

- To the assembler, labels are just labels – no difference
- To programmer, it’s wise to distinguish:
  - **Function entry point**: Meets calling conventions, usable by others
  - **Inner label**: Bookmark inside function for a loop/conditional
- Tip: Start all inner labels with an underscore to distinguish

```assembly
# takes a char* string, returns number of space characters
count_spaces:
   # we'll use the provided a0 as our moving pointer
   li $t1, 32        # value of a space character
   li $v0, 0         # running total starts at 0
   _loop:
      lb $t0, 0($a0)    # get this character
      beqz $t0, _endloop # break on null terminator
      bne $t0, $t1, _dontincrement # check if space
      addi $v0, $v0, 1    # if so, increment
   _dontincrement:
      addi $a0, $a0, 1    # increment pointer
      j _loop              # regardless, loop
   _endloop:
      jr $ra              # return
```
Keep your functions separate

- Functions should be **contiguous** with **one single entry point** and clearly defined return point(s)

```c
# takes a char* string, returns number of space characters
count_spaces:
    # we'll use the provided a0 as our moving pointer
    li $t1, 32
    li $v0, 0
_loop:
    lb $t0, 0($a0)
    beqz $t0, _endloop
    j _skip
send_email_to_grandma:
<code>              # wait.  
<code>              # what?      
<code>              # why is there another
jr $ra
_skip:
    bne $t0, $t1, _dontincrement # check if space
    addi $v0, $v0, 1 # if so, increment
_dontincrement:
    addi $a0, $a0, 1 # increment pointer
_j_loop
_endloop:
    jr $ra # return
```
Let’s walk through simple.s

- See simple.s, linked from course site by recitation 3
Factorial (skimming base case of recursion!)

\[
\text{fact: } \quad \text{addi } \$sp, \$sp, -8 \quad \# \text{ open frame (2 words)} \\
\text{sw } \$ra,4(\$sp) \quad \# \text{ save return address} \\
\text{sw } \$s0,0(\$sp) \quad \# \text{ save } \$s0
\]

\# handle base case (not real code here) 
\# if \$a0=1, set \$v0=1 and jump to _clean

\[
\text{move } \$s0, \$a0 \quad \# \text{ copy } \$a0 \text{ to } \$s0 \\
\text{addi } \$a0, \$a0, -1 \quad \# \text{ pass arg via } \$a0 \\
\text{jal fact} \quad \# \text{ recursive call} \\
\text{mul } \$v0, \$s0, \$v0 \quad \# \text{ value returned via } \$v0
\]

\_clean: \text{lw } \$s0,0(\$sp) \quad \# \text{ restore } \$s0 \\
\text{lw } \$ra,4(\$sp) \quad \# \text{ restore } \$ra \\
\text{addi } \$sp, \$sp, 8 \quad \# \text{ collapse frame} \\
\text{jr } \$ra \quad \# \text{ return, value in } \$v0
Summarizing the regions of memory

<table>
<thead>
<tr>
<th>C equivalent</th>
<th>Stack memory (locals)</th>
<th>Heap memory (malloc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>// outside of function int x = 10;</td>
<td>// inside a function int y;</td>
<td>int* p = malloc(5*sizeof(int));</td>
</tr>
</tbody>
</table>

### How to make?

- **Global memory (static)**
  - Declare labels and directives (.word, .space, .asciiz) in .data
  - .data
    - x: .word 10

- **Stack memory (locals)**
  - Bring down stack pointer $sp$ at start of func
    - addiu $sp$, $sp$, -4

- **Heap memory (malloc)**
  - use sbrk syscall (9)
    - li $v0, 9
    - li $a0, 20
    - syscall
    - move $s0, $v0 # keep ptr in $s0

### How to use?

- **Global memory (static)**
  - Use `la` to get the address associated with a label
    - `la $t0, x`
    - `lw $t1, ($t0)`

- **Stack memory (locals)**
  - use addresses at fixed offsets from $sp$
    - `lw $t0, 0($sp)`

- **Heap memory (malloc)**
  - Keep ahold of result pointer in a reg or in memory
    - `lw $t0, 0($s0)`

### How to free?

- **Global memory (static)**
  - No need! 😊

- **Stack memory (locals)**
  - raise up the stack pointer $sp$ at end of func
    - addiu $sp$, $sp$, 4

- **Heap memory (malloc)**
  - You’d implement malloc() and free() yourself, which we’re not doing in this class (safe to ignore for us) 😊

---

Yes, regular locals can live in the stack, but we usually just map them to registers instead. We only give them actual stack memory if we need a pointer to them. Instead, we primarily use the stack to save/restore registers to satisfy calling conventions.
Getting the words right

• My hypothesis: the way you phrase things can indicate good/bad understanding of this material

• Which of the phrases below make sense?

<table>
<thead>
<tr>
<th>Statement</th>
<th>Makes sense?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have an int in my register</td>
<td>Yes</td>
</tr>
<tr>
<td>I have a string in my register</td>
<td>No, but your register can point to a string</td>
</tr>
<tr>
<td>I have a float in my register</td>
<td>Yes, assuming it's a float register ($f$)</td>
</tr>
<tr>
<td>I have a struct in my register</td>
<td>No, but your register can point to a struct</td>
</tr>
<tr>
<td>My register points to an int</td>
<td>Yes</td>
</tr>
<tr>
<td>My register points to a string</td>
<td>Yes</td>
</tr>
<tr>
<td>My register points to a struct</td>
<td>Yes</td>
</tr>
<tr>
<td>68 bytes from pointer $t0$ is a place in a register</td>
<td>No, registers are of fixed, small size. They don't live in memory.</td>
</tr>
<tr>
<td>68 bytes from pointer $t0$ is a place in memory</td>
<td>Yes</td>
</tr>
</tbody>
</table>
All of MIPS in two pages

• Print this quick reference linked from the course page
Calling convention summary

- **Privacy:**
  - A function may not assume the state of any registers, except that $a$ registers have arguments and $ra$ has the return address. Put return value into $v$ register(s).

- **Callee-saved:**
  - A function may not leave $s$ registers in a modified state when returning.
  - At the top of a function, save any $s/ra$ registers that will be changed; restore right before returning.

- **Caller-saved:**
  - When making a call, save any $t$ registers you care about; restore right after it returns.
  - Minimize this by using $s$ registers in this case where possible.

- **Stack frame:**
  - At the top of a function, reserve space (decrementing $sp$) for any saving needed (for both $s/ra$ and $t$) as well as any local variables needing actual memory addresses as opposed to registers. Restore $sp$ before returning.
Outline

• What is an ISA?
• Assembly programming (in the MIPS ISA)
• Other ISAs
What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?

- **Implementability**
  - Easy to design high-performance implementations (i.e., microarchitectures)?

- **Compatibility**
  - Easy to maintain programmability as languages and programs evolve?
  - Easy to maintain implementability as technology evolves?
Programmability

- Easy to express programs efficiently?
  - For whom?

**Human**
- Want high-level coarse-grain instructions
  - As similar to HLL as possible
- This is the way ISAs were pre-1985
  - Compilers were terrible, most code was hand-assembled

**Compiler**
- Want low-level fine-grain instructions
  - Compiler can’t tell if two high-level idioms match exactly or not
- This is the way most post-1985 ISAs are
  - Optimizing compilers generate much better code than humans
  - ICQ: Why are compilers better than humans?
Implementability

- Every ISA can be implemented
  - But not every ISA can be implemented well
  - Bad ISA $\rightarrow$ bad microarchitecture (slow, power-hungry, etc.)

- We’d like to use some of these high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution
  - We’ll discuss these later in the semester

- Certain ISA features make these difficult
  - Variable length instructions
  - Implicit state (e.g., condition codes)
  - Wide variety of instruction formats
Compatibility

- Few people buy new hardware if it means they have to buy new software, too
  - Intel was the first company to realize this
  - ISA must stay stable, no matter what (microarch. can change)
    - x86 is one of the ugliest ISAs EVER, but survives
  - Intel then forgot this lesson: IA-64 (Itanium) was a new ISA*

- **Backward compatibility**: very important
  - New processors must support old programs (can’t drop features)

- **Forward (upward) compatibility**: less important
  - Old processors must support new programs
    - New processors only re-define opcodes that trapped in old ones
    - Old processors emulate new instructions in low-level software
RISC vs. CISC

- **RISC**: reduced-instruction set computer
  - Coined by Patterson in early 80’s (ideas originated earlier)
- **CISC**: complex-instruction set computer
  - Not coined by anyone, term didn’t exist before “RISC”

- Religious war (one of several) started in mid 1980’s
  - RISC (MIPS, Alpha, Power) “won” the technology battles
  - CISC (IA32 = x86) “won” the commercial war
    - Compatibility a stronger force than anyone (but Intel) thought
    - Intel beat RISC at its own game … more on this soon
The Setup

- Pre-1980
  - Bad compilers
  - Complex, high-level ISAs
  - Slow, complicated, multi-chip microarchitectures

- Around 1982
  - Advances in VLSI made single-chip microprocessor possible...
    - Speed by integration, on-chip wires much faster than off-chip
  - ...but only for very small, very simple ISAs
  - Compilers had to get involved in a big way

- RISC manifesto: create ISAs that...
  - Simplify single-chip implementation
  - Facilitate optimizing compilation
The RISC Tenets

- **Single-cycle execution (simple operations)**
  - CISC: many multi-cycle operations

- **Load/store architecture**
  - CISC: register-memory and memory-memory instructions

- **Few memory addressing modes**
  - CISC: many modes

- **Fixed instruction format**
  - CISC: many formats and lengths

- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance

**Summary**

1. Make it easy to implement in hardware
2. Make it easy for compiler to generate code
Intel 80x86 ISA (aka x86 or IA-32)

- Binary compatibility across generations
- 1978: 8086, 16-bit, registers have dedicated uses
- 1980: 8087, added floating point (stack)
- 1982: 80286, 24-bit
- 1985: 80386, 32-bit, new instrs → GPR almost
- 1989-95: 80486, Pentium, Pentium II
- 1997: Added MMX instructions (for graphics)
- 1999: Pentium III
- 2002: Pentium 4
- 2004: “Nocona” 64-bit extension (to keep up with AMD)
- 2006: Core2
- 2007: Core2 Quad
- 2013: Haswell – added transactional mem features
80x86 Registers, Addressing Modes, Instructions

- Eight 32-bit registers (not truly general purpose)
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
  - (Sixteen registers in modern 64-bit, plus several ‘weird’ registers)
- Six 16-bit registers for code, stack, & data
- 2-address ISA
  - One operand is both source and destination
- NOT a Load/Store ISA
  - One operand can be in memory
- Variable size instructions: 1-byte to 17-bytes, e.g.:
  - Jump (JE) 2-bytes
  - Push 1-byte
  - Add Immediate 5-bytes
How Intel Won Anyway

- x86 won because it was the first 16-bit chip by 2 years
  - IBM put it into its PCs because there was no competing choice
  - Rest is historical inertia and “financial feedback”
    - x86 is most difficult ISA to implement and do it fast but...
    - Because Intel (and AMD) sells the most processors...
    - It has the most money...
    - Which it uses to hire more and better engineers...
    - Which it uses to maintain competitive performance ...
    - And given equal performance compatibility wins...
    - So Intel (and AMD) sells the most processors...

- Moore’s law has helped Intel in a big way
  - Most engineering problems can be solved with more transistors
Current Approach: Pentium Pro and beyond

- Instruction decode logic translates into micro-ops
- Fixed-size instructions moving down execution path
- Execution units see only micro-ops
  + Faster instruction processing with backward compatibility
  + Execution unit as fast as RISC machines like MIPS
  - Complex decoding
  - We work with MIPS to keep decoding simple/clean
  - Learn x86 on the job!

Learn exactly how this all works in ECE 552 / CS 550
Concluding Remarks

1. Keep it simple and regular
   • Uniform length instructions
   • Fields always in same places

2. Keep it simple and fast
   • Small number of registers

3. Make the common case fast
   • Compromises inevitable \(\rightarrow\) there is no perfect ISA
Outline

• What is an ISA?
• Assembly programming (in the MIPS ISA)
• Other ISAs