ECE 550: Fundamentals of Computer Systems and Engineering

Instruction Set Architecture
MIPS

Admin

- Homework #2 Out now
  - Due October 2nd
  - VHDL part: more work than hwk1
- Reading
  - Chapter 2

Last time...

- Who can remind us what we did last time?

Last time...

- Who can remind us what we did last time?
  - Finite State Machines
  - Division

Now: Moving to software

- C is pretty low level
  - Most of you: in 551, can program in C
  - Others: assume you know programming, pointers, etc.
  - But compiled into assembly...
- Lower level programming
  - What does assembly look like?
  - How does software communicate with hardware?
  - What variations can there be?
  - Advantages/disadvantages?

Assembly

- Assembly programming:
  - 1 (sometimes two) machine instructions at a time
  - Still in "human readable form"
  - add r1, r2, r3
  - Much "lower level" than any other programming
  - Limited number of registers vs unlimited variables
  - Flat scope
    - (who can remind us what scope is? Hint: not mouthwash)
Registers

- Two places processors can store data
  - Registers (saw these—sort of):
    - In processor
    - Few of them (e.g., 32)
    - Fast (more on this much later in semester)
  - Memory (later):
    - Outside of processor
    - Huge (e.g., 4GB)
    - Slow (generally about 100—200x slower than registers, more later)
- For now: think of registers like "variables"
  - But only 32 of them
  - E.g., $r1 = r2 + r3$ much like $x = y + z$

Assembly too high level for machine

- Human readable is not (easily) machine executable
  - add $r1, r2, r3$
- Instructions are numbers too!
  - Bit fields (like FP numbers)
- Instruction Format
  - Establishes a mapping from "instruction" to binary values
  - Which bit positions correspond to which parts of the instruction (operation, operands, etc.)
- Assembler does this translation
  - Humans don’t typically need to write raw bits

What Must be Specified?

- Instruction "opcode"
  - What should this operation do? (add, subtract,...)
- Location of operands and result
  - Registers (which ones?)
  - Memory (what address?)
  - Immediates (what value?)
- Data type and Size
  - Usually included in opcode
  - E.g., signed vs unsigned int (if it matters)
- What instruction comes next?
  - Sequentially next instruction, or jump elsewhere

The ISA

- Instruction Set Architecture (ISA)
  - Contract between hardware and software
    - Specifies everything hardware and software need to agree on
      - Instruction encoding and effects
      - Memory endian-ness
      - (lots of other things that won’t make sense yet)
- Many different ISAs
  - x86 and x86_64 (Intel and AMD)
  - POWER (IBM)
  - MIPS
  - ARM
  - SPARC (Oracle)

Our focus: MIPS

- We will work with MIPS
  - x86 is ugly (x86_64 is less ugly, but still nasty)
  - MIPS is relatively “clean”
    - More on this in a minute

But I don’t have a MIPS computer?

- We’ll be using SPIM
  - Command line version: spim
  - Graphical version: xspim
- Edit in emacs, run in SPIM
ISAs: RISC vs CISC

- Two broad categories of ISAs:
  - Complex Instruction Set Computing
    - Came first, days when people always directly wrote assembly
    - Big complex instructions
  - Reduced Instruction Set Computing
    - Goal: make hardware simple and fast
    - Write in high level language, let compiler do the dirty work
      - Rely on compiler to optimize for you
  - Note:
    - Sometimes fuzzy: ISAs may have some features of each
    - Common mis-conception: not about how many different insns!

Reduction Instruction Set Computing

- Simple, fixed length instruction encoding
- Few memory addressing modes
- Instructions have one effect
- "Many" registers (e.g., 32)
- Three-operand arithmetic (dest = src1 op src2)
- Load-store ISA

Addressing Modes

- Memory location: how to specify address
  - Simple (RISCy)
    - Register + Immediate (e.g., address = r4 + 16)
    - Register + Register (e.g., address = r4 + r7)
  - Complex (CISCy)
    - Auto-increment (e.g., address = r4; r4 = r4 + 4)
    - Scaled Index (e.g., address = r4 + (r2 << 2) + 0x1234500)
    - Memory indirect (e.g., address = memory[r4])

Load-Store ISA

- Load-store ISA:
  - Specific instructions (loads/stores) to access memory
    - Loads read memory (and only read memory)
    - Stores write memory (and only write memory)
  - Contrast with
    - General memory operands (r4 = mem[r5] + r3)
    - Memory/memory operations: mem[r4] = mem[r5] + r3

Memory Addressing Modes

- Memory location: how to specify address
  - Simple (RISCy)
    - Register + Immediate (e.g., address = r4 + 16)
    - Register + Register (e.g., address = r4 + r7)
  - Complex (CISCy)
    - Auto-increment (e.g., address = r4; r4 = r4 + 4)
    - Scaled Index (e.g., address = r4 + (r2 << 2) + 0x1234500)
    - Memory indirect (e.g., address = memory[r4])
Stored Program Computer

- Instructions: a fixed set of built-in operations
- Instructions and data are stored in memory
  - Allows general purpose computation!
- Fetch-Execute Cycle
  ```java
  while (!done)
    fetch instruction
    execute instruction
  ```
- Effectively what hardware does
- This is what the SPIM Simulator does

How are Instructions Executed?

- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the #s)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

More Details on Execution?

- Previous slides high level overview
  - Called von Neumann model
  - John von Neumann: Eniac
- More details: How hardware works
  - Late September
- Now, diving into assembly programming/MIPS

Assembly Programming

- How do you write an assembly program?
- How do you write a program (in general)?

5 Step Plan (ECE 551)

- 5 Steps to write any program:
  1. Work an example yourself
  2. Write down what you did
  3. Generalize steps from 2
  4. Test generalized steps on different example
  5. Translate generalized steps to code

How to Write a Program

- How I teach programming:
  1. Work an example yourself
  2. Write down what you did
  3. Generalize steps from 2
  4. Test generalized steps on different example
  5. Translate generalized steps to code
    - Then translate to lower level language
Why do I bring this up?

- Very Hard:
  - Problem
  - Correctly Working Assembly

- Easier:
  - Problem
  - Algorithm
  - High-level Language Implementation
  - Correctly Working Assembly

Our focus

- We will focus on the assembly step
  - Assume you know how to devise an algorithm for a problem
  - I’ll use C.

Simplest Operations We Might Want?

- What is the simplest computation we might do?

Simplest Operations We Might Want?

- What is the simplest computation we might do?
  - Add two numbers:
    \[ x = a + b; \]

Simplest Operations We Might Want?

- What is the simplest computation we might do?
  - Add two numbers:
    \[ x = a + b; \]

MIPS Integer Registers

- Recall: registers
  - Fast
  - In CPU
  - Directly compute on them
  - 31 x 32-bit GPRs ($R0 = 0$)
  - Also FP registers
  - A few special purpose regs too
    - PC = Address of next insn
  
<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r0$</td>
<td>0</td>
</tr>
<tr>
<td>$r1$</td>
<td></td>
</tr>
<tr>
<td>$r2$</td>
<td></td>
</tr>
<tr>
<td>$r3$</td>
<td></td>
</tr>
<tr>
<td>$r31$</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
</tr>
<tr>
<td>lo</td>
<td></td>
</tr>
<tr>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>
Add reads its source registers, and uses their values directly ("register direct")

You all do the next instruction!
### Executing Add

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td><code>add $r8, $r4, $r6</code></td>
<td>R0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td><code>add $r5, $r8, $r7</code></td>
<td>R1</td>
<td>1234 5678</td>
</tr>
<tr>
<td>1008</td>
<td><code>add $r6, $r6, $r6</code></td>
<td>R2</td>
<td>C001 0000</td>
</tr>
<tr>
<td>100C</td>
<td></td>
<td>R3</td>
<td>1BAD 0000</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>R4</td>
<td>9999 9998</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R5</td>
<td>9999 999C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R6</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R7</td>
<td>0000 0002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R8</td>
<td>9999 999A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R9</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R10</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R11</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R12</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC</td>
<td>0000 1008</td>
</tr>
</tbody>
</table>

Its perfectly fine to have $r6 as a src and a dst
This is just like `x = x + x;` in C, Java, etc:

\[ 1 + 1 = 2 \]

### MIPS Instruction Formats

**R-type: Register-Register**

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00010</td>
<td>00011</td>
<td>00001</td>
<td>000000</td>
<td>100000</td>
</tr>
</tbody>
</table>

**I-type: Register-Immediate**

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00010</td>
<td>00011</td>
<td>00001</td>
</tr>
</tbody>
</table>

**J-type: Jump / Call**

<table>
<thead>
<tr>
<th>Op</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
</tr>
</tbody>
</table>

Terminology

- **Op** = opcode
- **Rs, Rt, Rd** = register specifier

### Executing Add

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td><code>add $r8, $r4, $r6</code></td>
<td>R0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td><code>add $r5, $r8, $r7</code></td>
<td>R1</td>
<td>1234 5678</td>
</tr>
<tr>
<td>1008</td>
<td><code>add $r6, $r6, $r6</code></td>
<td>R2</td>
<td>C001 0000</td>
</tr>
<tr>
<td>100C</td>
<td></td>
<td>R3</td>
<td>1BAD 0000</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>R4</td>
<td>9999 9998</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R5</td>
<td>9999 999C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R6</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R7</td>
<td>0000 0002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R8</td>
<td>9999 999A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R9</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R10</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R11</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R12</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC</td>
<td>0000 100C</td>
</tr>
</tbody>
</table>

Its perfectly fine to have $r6 as a src and a dst
This is just like `x = x + x;` in C, Java, etc:

\[ 1 + 1 = 2 \]
Other Similar Instructions

- sub $rDest, $rSrc1, $rSrc2
- mul $rDest, $rSrc1, $rSrc2 (pseudo-instruction)
- div $rDest, $rSrc1, $rSrc2 (pseudo-instruction)
- add $rDest, $rSrc1, $rSrc2
- or $rDest, $rSrc1, $rSrc2
- xor $rDest, $rSrc1, $rSrc2
- ...

End of Appendix B: listing of all instructions
  - Good reference, don’t need to read every insn
  - Will provide insn reference on tests

Pseudo Instructions

- Some "instructions" are pseudo-instructions
  - Actually assemble into 2 instructions:
    - mul $r1, $r2, $r3 is really
      - mul $r2, $r3
      - mflo $r1
  - mul takes two srcs, writes special regs lo and hi
  - mflo moves from lo into dst reg

What if I want to add a constant?

- Suppose I need to do
  - $x = x + 1$
    - Idea one: Put 1 in a register, use add
    - Problem: How to put 1 in a register?
    - Idea two: Have instruction that adds an immediate
      - Note: also solves problem in idea one

I-Type <op> rt, rs, immediate

- Immediate: 16 bit value

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>001000</td>
<td>000</td>
<td>000001011000</td>
</tr>
</tbody>
</table>

  Add Immediate Example
  - addi $t1, $t2, 100

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs</th>
<th>Rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>001000</td>
<td>000</td>
<td>000001011000</td>
</tr>
</tbody>
</table>

Using addi to put a const in register

- Can use addi to put a constant into a register:
  - $x = 42$
  - Can be done with
    - addi $r7, $r0, 42
  - Because $r0 is always 0.

- Common enough it has its own pseudo-insn:
  - li $r7, 42
  - Stands for load immediate, works for 16-bit immediate

Many insns have Immediate Forms

- Add is not the only one with an immediate form
  - andi, ori, xori,sll,sr,sra,...
- No subi
  - Why not?
- No muli or divi
  - Though some ISAs have them
Assembly programming something

- Consider the following C fragment:
  ```c
  int tempF = 87;
  int a = tempF - 32;
  a = a * 5;
  int tempC = a / 9;
  ```
  - Let's write assembly for it
  - First, need registers for our variables:
    - `tempF` = `$r3`
    - `a` = `$r4`
    - `tempC` = `$r5`
  - Now, give it a try (use `$r6`, `$r7`,... as temps if you need)...

```assembly
li $r3, 87
addi $r4, $r3, -32
li $r6, 5
mul $r4, $r4, $r6
li $r6, 9
div $r5, $r4, $r6
```

Accessing Memory

- MIPS is a "load-store" ISA
  - Who can remind us what that means?

I-Type <op> rt, rs, immediate

- Load Word Example
  ```assembly
  lw $1, 100($2)      # $1 = Mem[$2+100]
  ```
  - I-type: Register-Immediate
  - Data sizes / types
    - Loads and Stores come in multiple sizes
    - Reflect different data types
    - The `w` in lw/sw stands for "word" (= 32 bits)
    - Can also load bytes (8 bits), half words (16 bits)
    - Smaller sizes have signed/unsigned forms
    - See Appendix B for all variants
C and assembly: loads/stores

```c
int x; # in r1
int * p; # in r2
int ** q; # in r3
... ...

x = *p;
**q = x;
w $r2, 0($r3)

p = *q;
w $r2, 0($r3)
p[4] = x;
sw $r1, 16($r2)
```

Executing Memory Ops

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>lw $r1, 0($r2)</td>
</tr>
<tr>
<td>1004</td>
<td>lw $r4, 4($r3)</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r1, 8($r4)</td>
</tr>
<tr>
<td>100C</td>
<td>lw $r2, 0($r3)</td>
</tr>
</tbody>
</table>
| 1010    | ...

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R1</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R2</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R3</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R4</td>
<td>9999 9999</td>
</tr>
<tr>
<td>R5</td>
<td>9999 9999</td>
</tr>
<tr>
<td>R6</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R7</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R8</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R9</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R10</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R11</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R12</td>
<td>0000 0000</td>
</tr>
<tr>
<td>PC</td>
<td>0000 0010</td>
</tr>
</tbody>
</table>

ISA/MIPS

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>lw $r1, 0($r2)</td>
</tr>
<tr>
<td>1004</td>
<td>lw $r4, 4($r3)</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r1, 8($r4)</td>
</tr>
<tr>
<td>100C</td>
<td>lw $r2, 0($r3)</td>
</tr>
</tbody>
</table>
| 1010    | ...

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R1</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R2</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R3</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R4</td>
<td>9999 9999</td>
</tr>
<tr>
<td>R5</td>
<td>9999 9999</td>
</tr>
<tr>
<td>R6</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R7</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R8</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R9</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R10</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R11</td>
<td>0000 0000</td>
</tr>
<tr>
<td>R12</td>
<td>0000 0000</td>
</tr>
<tr>
<td>PC</td>
<td>0000 0010</td>
</tr>
</tbody>
</table>

ISA/MIPS
Making control decision

- Control constructs—decide what to do next:
  
  ```
  if (x == y) {
      ...
  } else {
      ...
  }
  ... while (z < q) {
      ...
  }
  ```


The Program Counter (PC)

- Special register (PC) that points to instructions
- Contains memory address (like a pointer)
- Instruction fetch is
  
  ```
  inst = mem[pc]
  ```
- So far, have fetched sequentially: PC = PC + 4
- Assumes 4 byte insns
- True for MIPS
- X86: variable size (nasty)
- May want to specify non-sequential fetch
  
  ```
  Change PC in other ways
  ```

I-Type <op> rt, rs, immediate

- PC relative addressing
- Branch Not Equal Example
  
  ```
  bne $1, $2, 100  # If ($1 != $2) goto [PC+4+400]
  ```

MIPS Compare and Branch

- Compare and Branch
  
  ```
  beq rs, rt, offset
  ```
- Compare to zero and Branch
  
  ```
  blez rs, offset
  ```
  ```
  bltz rs, offset
  ```
  ```
  bgtz rs, offset
  ```
  ```
  bgez rs, offset
  ```
- Also pseudo-insns for unconditional branch (b)

MIPS jump, branch, compare

- Inequality to something other than 0: require 2 insns
  
  ```
  Cond. set reg, branch if not zero or if zero
  ```
- ```
  slt $1,$2,$3  $1 = ($2 < $3) ? 1 : 0
  ```
- ```
  Compare less than; signed 2's comp.
  ```
- ```
  slti $1,$2,100  $1 = ($2 < 100) ? 1 : 0
  ```
- ```
  Compare < constant; signed 2's comp.
  ```
- ```
  sltu $1,$2,$3  $1 = ($2 < $3) ? 1 : 0
  ```
- ```
  Compare less than; unsigned
  ```
- ```
  sltiu $1,$2,100 $1 = ($2 < $3) ? 1 : 0 $1 = 0
  ```
- ```
  Compare < constant; unsigned
  ```
- ```
  beqz $1,100  if ($1 == $2) go to PC+4+400
  ```
- ```
  Branch if equal to 0
  ```
- ```
  bnez $1,100  if ($1 != $2) go to PC+4+400
  ```

Signed v.s. Unsigned Comparison

- R1= 0...00 0000 0000 0000 0000 0001
- R2= 0...00 0000 0000 0000 0000 0010
- R3= 1...11 1111 1111 1111 1111 1111
- After executing these instructions:
  
  ```
  slt r4,r2,r1
  ```
  ```
  slt r5,r3,r1
  ```
  ```
  slt r6,r2,r1
  ```
  ```
  slt r7,r3,r1
  ```
- What are values of registers r4 - r7? Why?
  
  ```
  r4 =  ; r5 =  ; r6 =  ; r7 =  
  ```
Signed v.s. Unsigned Comparison

- R1 = 0...00 0000 0000 0000 0001
- R2 = 0...00 0000 0000 0000 0010
- R3 = 1...11 1111 1111 1111 1111
- After executing these instructions:
  - slt r4,r2,r1
  - slt r5,r3,r1
  - sltu r6,r2,r1
  - sltu r7,r3,r1
- What are values of registers r4 - r7? Why?
  - r4 = 0; r5 = 1; r6 = 0; r7 = 0;

C and Assembly with branches

```c
int x; //assume x in r1
int y; //assume y in r2
int z; //assume z in r3
... ...
if (x != y) {
  z = z + 2;
  addi $r3, $r3, 2
} else {
  z = z - 4;
  addi $r3, $r3, -4
}
```

Labels

- Counting insns?
  - Error prone
  - Tricky: pseudo-insns
  - Un-maintainable
- Better: let assembler count
  - Use a label
  - Symbolic name for target
  - Assembler computes offset

J-Type <op> immediate

- 16-bit imm limits to +/- 32K insns
- Usually fine, but sometimes need more...
- J-type insns provide long range, unconditional jump:
  - J-type: Jump / Call
  - Specifications lowest 28 bits of PC
    - Upper 4 bits unchanged
    - Range: 64 Million instruction (256 MB)
- Can jump anywhere with jr $reg (jump register)

Remember our F2C program fragment?

- Consider the following C fragment:
  - int tempF = 87;
  - int a = tempF - 32;
  - a = a * 5;
  - int tempC = a / 9;
  - If we were really doing this... div $r5, $r4, $r6
  - We would write a function to convert f2c and call it
More likely: a function

- Like this:
  ```c
  int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
  }
  ... 
  int tempC = f2c (87);
  ```

More likely: a function to convert

- Like this:
  ```c
  int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
  }  
  int tempC = f2c (87);  //jal f2c
  ```

- But that's not all...

Need a way to call f2c and return

- Call: Jump... but also remember where to go back
  - May be many calls to f2c() in the program
  - Need some way to know where we were
  - Instruction for this jal
    - jal label
      - Store PC +4 into register $31
      - Jump to label
  - Return: Jump... back to wherever we were
    - Instruction for this jr
      - jr $31
      - Jump back to address stored by jal in $31

More likely: a function to convert

- Like this:
  ```c
  int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
  }
  ... 
  int tempC = f2c (87);  //jal f2c
  ```

- Need to pass 87 as argument to f2c

More likely: a function to convert

- Like this:
  ```c
  int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
  }
  ... 
  int tempC = f2c (87);  //jal f2c
  ```

- Also, may want to use same registers in multiple functions
  - What if f2c called something? Would re-use $31
Callings Convention

- All of these are reasons for a calling convention
  - Agreement of how registers are used
  - Where arguments are passed, results returned
  - Who must save what if they want to use it
  - Etc..

- Alternative: inter-procedural register allocation
  - More work for compiler
  - Only know one real compiler that does this

MIPS Register Naming Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero, constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at, reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0, return result</td>
</tr>
<tr>
<td>3</td>
<td>v1, (can be used as temp)</td>
</tr>
<tr>
<td>4</td>
<td>a0, Argument passing</td>
</tr>
<tr>
<td>5</td>
<td>a1, Argument passing</td>
</tr>
<tr>
<td>6</td>
<td>a2, reserved for OS</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0, temporaries</td>
</tr>
<tr>
<td>9</td>
<td>t1</td>
</tr>
<tr>
<td>10</td>
<td>t2</td>
</tr>
<tr>
<td>11</td>
<td>t3</td>
</tr>
<tr>
<td>12</td>
<td>t4</td>
</tr>
<tr>
<td>13</td>
<td>t5</td>
</tr>
<tr>
<td>14</td>
<td>t6</td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0, Callee saves</td>
</tr>
<tr>
<td>17</td>
<td>s1</td>
</tr>
<tr>
<td>18</td>
<td>s2</td>
</tr>
<tr>
<td>19</td>
<td>s3</td>
</tr>
<tr>
<td>20</td>
<td>s4</td>
</tr>
<tr>
<td>21</td>
<td>s5</td>
</tr>
<tr>
<td>22</td>
<td>s6</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>b8, Caller saves (continued)</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>

Caller Saves

- Caller saves registers
  - If some code is about to call another function...
  - And it needs the value in a caller saves register ($t0,$t1...)
  - Then it has to save it on the stack before the call
  - And restore it after the call

Callee Saves

- Callee saves registers
  - If some code wants to use a callee saves register (at all)
  - It has to save it to the stack before it uses it
  - And restore it before it returns to its caller
  - But, it can assume any function it calls will not change the register
    - Either won't use it, or will save/restore it

More likely: a function to convert

```c
int f2c (int tempF) {
    int a = tempF - 32;
    int tempC = a / 9;
    return tempC;
}
```
More likely: a function to convert

```c
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```asm
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
```

A smart compiler would just do
```
    div $v0, $s0, $t1
```

ECE 550 (Hilton): ISA/MIPS

More likely: a function to convert

```c
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```asm
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
```

```
    addi $v0, $t2, 0
    jr $ra
    int tempC = f2c(87)  ...
    addi $a0, $s0, 87
    jal f2c
```

ECE 550 (Hilton): ISA/MIPS

More likely: a function to convert

```c
int f2c (int tempF) {
    int a = tempF - 32;
    a = a * 5;
    int tempC = a / 9;
    return tempC;
}
```

```asm
f2c:
    addi $t0, $a0, -32
    li $t1, 5
    mul $t0, $t0, $t1
    li $t1, 9
    div $t2, $t0, $t1
    addi $v0, $t2, 0
    jr $ra
```

```
    addi $v0, $t2, 0
    jr $ra
    int tempC = f2c(87)  ...
    addi $a0, $s0, 87
    jal f2c
```

ECE 550 (Hilton): ISA/MIPS
What it would take to make SPIM

```
.globl f2c  # f2c can be called from any file
.ent f2c  # entry point of function
.text    # goes in "text" region
.f2c:    # (remember memory picture?)
    addi $t0, $a0, -32
    li    $t1, 5
    mul   $t0, $t0, $t1
    li    $t1, 9
    div   $t2, $t0, $t1
    addi $v0, $t2, 0
    jr    $ra
.end f2c  # end of this function
```

Assembly Language (cont.)

- Directives: tell the assembler what to do...
- Format "."<string> [arg1], [arg2] ...

Examples
- .data [address] # start a data segment.
- # [optional beginning address]
- .text [address] # start a code segment.
- .align n # align segment on 2^n byte boundary.
- .ascii <string> # store a string in memory.
- .asciiz <string> # store a null terminated string in memory
- .word w1, w2, . . . , wn # store n words in memory.

The Stack

- May need to use the stack for...
  - Local variables whose address is taken
    - (Can’t have “address of register”)
  - Saving registers
    - Across calls
    - Spilling variables (not enough regs)
  - Passing more than 4 arguments

Stack Layout

- Stack is in memory:
  - Use loads and stores to access
  - But what address to load/store?
- Two registers for stack:
  - Stack pointer ($sp): Points at end (bottom) of stack
  - Frame pointer ($fp): Points at top of current stack frame

Call-Return Linkage: Stack Frames

MIPS/GCC Procedure Calling

- Calling Procedure
- Step-1: Setup the arguments:
  - The 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: Cleanup stack (if more than 4 args)
MIPS/GCC Procedure Calling

- Called Routine (if any frame is needed)
- Step-1: Establish stack frame.
  - Subtract the frame size from the stack pointer.
  - Addiu $sp, $s0, <frame-size>
  - Typically, minimum frame size is 32 bytes (8 words).
- Step-2: Save callee saved registers in the frame.
  - Register $fp is always saved.
  - Register $ra is saved if routine makes a call.
  - Registers $s0-$s7 are saved if they are used.
- Step-3: Establish Frame pointer
  - Add the stack <frame size> - 4 to the address in $sp
  - Addiu $fp, $sp, <frame-size> - 4

Step-4: Return

Just did jal 1000

Addr
1000
1004
1008
100C
1010
1014
1018
101C
1020
1024
1028
102C
1030
1034
Value
$0  0000 0000
$s  0000 0000
$v  0000 0000
$v1 0000 0000
$v2 0000 0000
$v3 0000 0000
$v4 0000 0000
$v5 0000 0000
$v6 0000 0000
$v7 0000 0000
$fp 0000 0000
$ra 0000 0000

Addr
FFFO 0001 0070
FFEC 1234 5678
FFEB 9999 9999
FFE4 0000 2568
FFEO 0001 0040
FFDC
FFDA
FFDB
FFCC
FFCB
FFCD
FFBC

Allocation space on stack: $0 0000 2348
PC 0000 1004

ECE 550 (Hilton): ISA/MIPS

Execution example: Calling with frames
Execution example: Calling with frames

Addr | Instruction | Reg | Value | Addr | Value
--- | --- | --- | --- | --- | ---
1000 | subiu $sp, $sp, 16 | $r0 | 0000 0000 | FFFD | 0000 0000
1004 | sw $f0, 0($sp) | $at | 0000 0000 | FFFC | 1234 5678
1008 | sw $ra, 4($sp) | $v0 | 4242 4242 | FFFE | 9999 9999
100C | sw $sp, 8($sp) | $v1 | 0000 8010 | FFFB | 0000 2348
1010 | addi $f0, $sp, 12 | $a0 | 0000 1234 | FFED | 0000 0000
1014 | add $s0, $a0, $a1 | $a1 | 5678 0001 | FFNF |
1018 | jal 4200 | $a2 | 0000 0000 | FFD0 | 0000 1008
101C | add $t0, $v0, $s0 | $a3 | 0000 0000 | FFD1 | 0000 1008
1020 | lw $v0, 4($t0) | $t0 | 0000 0000 | FFD2 | 0000 1234
1024 | lw $s0, -4($fp) | $t1 | 0000 0000 | FFD3 | 0000 1234
1028 | lw $ra, -8($fp) | $t2 | 0000 0000 | FFD4 | 0000 2348
102C | lw $f0,-12($fp) | $t3 | 0000 0000 | FFD5 | 0000 2348
1030 | addi $sp, $sp, 16 | $sp | 0000 FFDD | FF6B | FFFD |
1034 | jr $ra | $fp | 0000 0000 | FFF6 |

Save $ra
$r0 | 0000 2348
$ra | 0000 2348
PC | 0000 1018

Execution example: Calling with frames
### Execution example: Calling with frames

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>subiu $s0, $s0, 16</td>
<td>$r0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td>sw $f0, 0($s8)</td>
<td>$at</td>
<td>???? ????</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r4, 4($s0)</td>
<td>$v0</td>
<td>8675 3090</td>
</tr>
<tr>
<td>100C</td>
<td>sw $s0, 8($s0)</td>
<td>$v1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1010</td>
<td>addiu $f0, $s0, 12</td>
<td>$a0</td>
<td>???? ????</td>
</tr>
<tr>
<td>1014</td>
<td>add $s0, $a0, $a1</td>
<td>$s1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1018</td>
<td>jal 4200</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>101C</td>
<td>add $t0, $v0, $s0</td>
<td>$a2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1020</td>
<td>lw $v0, 0($t0)</td>
<td>$v2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1024</td>
<td>lw $lo, -4($fp)</td>
<td>$a3</td>
<td>???? ????</td>
</tr>
<tr>
<td>1028</td>
<td>lw $ra, -8($fp)</td>
<td>$t1</td>
<td>???? ????</td>
</tr>
<tr>
<td>102C</td>
<td>lw $fp, -12($fp)</td>
<td>$a4</td>
<td>???? ????</td>
</tr>
<tr>
<td>1030</td>
<td>addiu $s0, $s0, 16</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>1034</td>
<td>jr $ra</td>
<td>$v3</td>
<td>???? ????</td>
</tr>
</tbody>
</table>

Other function can do what it wants to the regs as it computes.

And make a stack frame Of its own.

---

### Execution example: Calling with frames

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>subiu $s0, $s0, 16</td>
<td>$r0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td>sw $f0, 0($s8)</td>
<td>$at</td>
<td>???? ????</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r4, 4($s0)</td>
<td>$v0</td>
<td>8675 3090</td>
</tr>
<tr>
<td>100C</td>
<td>sw $s0, 8($s0)</td>
<td>$v1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1010</td>
<td>addiu $f0, $s0, 12</td>
<td>$a0</td>
<td>???? ????</td>
</tr>
<tr>
<td>1014</td>
<td>add $s0, $a0, $a1</td>
<td>$s1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1018</td>
<td>jal 4200</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>101C</td>
<td>add $t0, $v0, $s0</td>
<td>$a2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1020</td>
<td>lw $v0, 0($t0)</td>
<td>$v2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1024</td>
<td>lw $lo, -4($fp)</td>
<td>$a3</td>
<td>???? ????</td>
</tr>
<tr>
<td>1028</td>
<td>lw $ra, -8($fp)</td>
<td>$t1</td>
<td>???? ????</td>
</tr>
<tr>
<td>102C</td>
<td>lw $fp, -12($fp)</td>
<td>$a4</td>
<td>???? ????</td>
</tr>
<tr>
<td>1030</td>
<td>addiu $s0, $s0, 16</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>1034</td>
<td>jr $ra</td>
<td>$v3</td>
<td>???? ????</td>
</tr>
</tbody>
</table>

But before it returns, it is responsible for restoring certain registers.

Including $sp and $f1, and $v0.

Value returned in $v0.

---

### Execution example: Calling with frames

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>subiu $s0, $s0, 16</td>
<td>$r0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td>sw $f0, 0($s8)</td>
<td>$at</td>
<td>???? ????</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r4, 4($s0)</td>
<td>$v0</td>
<td>8675 3090</td>
</tr>
<tr>
<td>100C</td>
<td>sw $s0, 8($s0)</td>
<td>$v1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1010</td>
<td>addiu $f0, $s0, 12</td>
<td>$a0</td>
<td>???? ????</td>
</tr>
<tr>
<td>1014</td>
<td>add $s0, $a0, $a1</td>
<td>$s1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1018</td>
<td>jal 4200</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>101C</td>
<td>add $t0, $v0, $s0</td>
<td>$a2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1020</td>
<td>lw $v0, 0($t0)</td>
<td>$v2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1024</td>
<td>lw $lo, -4($fp)</td>
<td>$a3</td>
<td>???? ????</td>
</tr>
<tr>
<td>1028</td>
<td>lw $ra, -8($fp)</td>
<td>$t1</td>
<td>???? ????</td>
</tr>
<tr>
<td>102C</td>
<td>lw $fp, -12($fp)</td>
<td>$a4</td>
<td>???? ????</td>
</tr>
<tr>
<td>1030</td>
<td>addiu $s0, $s0, 16</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>1034</td>
<td>jr $ra</td>
<td>$v3</td>
<td>???? ????</td>
</tr>
</tbody>
</table>

Do some more compuation (load addr not pictured).

---

### Execution example: Calling with frames

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>subiu $s0, $s0, 16</td>
<td>$r0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td>sw $f0, 0($s8)</td>
<td>$at</td>
<td>???? ????</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r4, 4($s0)</td>
<td>$v0</td>
<td>8675 3090</td>
</tr>
<tr>
<td>100C</td>
<td>sw $s0, 8($s0)</td>
<td>$v1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1010</td>
<td>addiu $f0, $s0, 12</td>
<td>$a0</td>
<td>???? ????</td>
</tr>
<tr>
<td>1014</td>
<td>add $s0, $a0, $a1</td>
<td>$s1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1018</td>
<td>jal 4200</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>101C</td>
<td>add $t0, $v0, $s0</td>
<td>$a2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1020</td>
<td>lw $v0, 0($t0)</td>
<td>$v2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1024</td>
<td>lw $lo, -4($fp)</td>
<td>$a3</td>
<td>???? ????</td>
</tr>
<tr>
<td>1028</td>
<td>lw $ra, -8($fp)</td>
<td>$t1</td>
<td>???? ????</td>
</tr>
<tr>
<td>102C</td>
<td>lw $fp, -12($fp)</td>
<td>$a4</td>
<td>???? ????</td>
</tr>
<tr>
<td>1030</td>
<td>addiu $s0, $s0, 16</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>1034</td>
<td>jr $ra</td>
<td>$v3</td>
<td>???? ????</td>
</tr>
</tbody>
</table>

Restore registers to return.

---

### Execution example: Calling with frames

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>subiu $s0, $s0, 16</td>
<td>$r0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td>sw $f0, 0($s8)</td>
<td>$at</td>
<td>???? ????</td>
</tr>
<tr>
<td>1008</td>
<td>sw $r4, 4($s0)</td>
<td>$v0</td>
<td>8675 3090</td>
</tr>
<tr>
<td>100C</td>
<td>sw $s0, 8($s0)</td>
<td>$v1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1010</td>
<td>addiu $f0, $s0, 12</td>
<td>$a0</td>
<td>???? ????</td>
</tr>
<tr>
<td>1014</td>
<td>add $s0, $a0, $a1</td>
<td>$s1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1018</td>
<td>jal 4200</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>101C</td>
<td>add $t0, $v0, $s0</td>
<td>$a2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1020</td>
<td>lw $v0, 0($t0)</td>
<td>$v2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1024</td>
<td>lw $lo, -4($fp)</td>
<td>$a3</td>
<td>???? ????</td>
</tr>
<tr>
<td>1028</td>
<td>lw $ra, -8($fp)</td>
<td>$t1</td>
<td>???? ????</td>
</tr>
<tr>
<td>102C</td>
<td>lw $fp, -12($fp)</td>
<td>$a4</td>
<td>???? ????</td>
</tr>
<tr>
<td>1030</td>
<td>addiu $s0, $s0, 16</td>
<td>$sp</td>
<td>???? ????</td>
</tr>
<tr>
<td>1034</td>
<td>jr $ra</td>
<td>$v3</td>
<td>???? ????</td>
</tr>
</tbody>
</table>

Restore registers to return.
Execution example: Calling with frames

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Reg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>subi $sp, $sp, 16</td>
<td>$s0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1004</td>
<td>sw $s0, 0($sp)</td>
<td>$at</td>
<td>???? ????</td>
</tr>
<tr>
<td>1008</td>
<td>sw $s1, 4($sp)</td>
<td>$v0</td>
<td>0001 0002</td>
</tr>
<tr>
<td>100C</td>
<td>sw $s2, 8($sp)</td>
<td>$v1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1010</td>
<td>addiu $s0, $s0, 12</td>
<td>$a0</td>
<td>???? ????</td>
</tr>
<tr>
<td>1014</td>
<td>add $s0, $a0, $a1</td>
<td>$s1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1018</td>
<td>jal 4200</td>
<td>$s2</td>
<td>???? ????</td>
</tr>
<tr>
<td>1020</td>
<td>lw $s0, 4($s0)</td>
<td>$a3</td>
<td>???? ????</td>
</tr>
<tr>
<td>1024</td>
<td>lw $s1, -4($sp)</td>
<td>$t0</td>
<td>DCED 42C5</td>
</tr>
<tr>
<td>102C</td>
<td>lw $s2, -8($fp)</td>
<td>$t1</td>
<td>???? ????</td>
</tr>
<tr>
<td>1030</td>
<td>addiu $s0, $s0, 16</td>
<td>$s0</td>
<td>0042 0420</td>
</tr>
<tr>
<td>1034</td>
<td>jr $ra</td>
<td>$fp</td>
<td>0000 FF00</td>
</tr>
<tr>
<td>1038</td>
<td>Restore registers to return</td>
<td>$fa</td>
<td>0000 2348</td>
</tr>
</tbody>
</table>

ECE 550 (Hilton): ISA/MIPS

Assembly Writing Tips and Advice

- Write C first, translate C -> Assembly
- One function at a time
- Pick registers for each variable
  - Must be in memory? Give it a stack slot (refer to by $fp+num)
  - Live across a call? Use an $s register
  - Otherwise, a $t
- Write prolog
  - Save raft (if needed)
  - Save any $s registers you use
- Translate line by line
- Write epilog

Why do we need FP?

- The frame pointer is not always required
  - May be able to get away without it
- When/why do we need it?
  - Debugging tools (gdb) use it to find frames
  - If you have variable length arrays
    - Stack pointer changes by amount not know at compile time
    - Variables still at constant offset from frame pointer
  - Good practice for this class to use it
    - Don’t prematurely optimize
System Call Instruction

- System call is used to communicate with the operating system, and request services (memory allocation, I/O).
- Load system call code (value) into Register $v0.
- Load arguments (if any) into registers $a0, $a1 or $f12 (for floating point).
- do: syscall
- Results returned in registers $v0 or $f0.
- Note: $v0 = $2, $a0=$4, $a1 = $5

SPIM System Call Support

- code service Arguments Result
- 1 print int $a0
- 2 print float $f12
- 3 print double$f12
- 4 print string $a0 (string address)
- 5 read integer integer in $v0
- 6 read float float in $f0
- 7 read double double in $f0
- 8 read string $a0=buffer, $a1=length
- 9 sbrk $a0=amount address in $v0
- 10 exit

MIPS Assembly General Rules

- One instruction per line.
- Numbers are base-10 integers or Hex w/ leading 0x.
- Identifiers: alphanumeric, _, . string starting in a letter or _
- Labels: identifiers starting at the beginning of a line followed by ":".
- Comments: everything following # till end-of-line.
- Instruction format: Space and "," separated fields.
  - [Label:] <op> reg1, [reg2], [reg3]  [# comment]
  - [Label:] <op> reg1, offset(reg2)  [# comment]
  - .Directive [arg1], [arg2], ...

Summary

- MIPS ISA and Assembly Programming
  - We'll use SPIM (or xspim)
  - Have seen most basic instruction types
  - See Appendix B for full insn reference