DEAR VARIOUS PARENTS, GRANDPARENTS, CO-WORKERS, AND OTHER "NOT COMPUTER PEOPLE."

WE DON'T MAGICALLY KNOW HOW TO DO EVERYTHING IN EVERY PROGRAM. WHEN WE HELP YOU, WE'RE USUALLY JUST DOING THIS:

START

FIND A MENU ITEM OR BUTTON WHICH LOOKS RELATED TO WHAT YOU WANT TO DO.

OK

I CAN'T FIND ONE

I'VE TRIED THEM ALL

DO YOU HAVE YOU BEEN TRYING THIS FOR OVER HALF AN HOUR?

CLICK IT

NO

NO

I CAN'T FIND ONE

OK

YES

YES

DO IT WORK?

ASK SOMEONE FOR HELP OR GIVE UP?

YOU'RE DONE!

PLEASE PRINT THIS FLOWCHART OUT AND TAPE IT NEAR YOUR SCREEN. CONGRATULATIONS; YOU'RE NOW THE LOCAL COMPUTER EXPERT!

http://www.xkcd.org/627/
“And in Review…”

- Memory is byte-addressable, but \texttt{lw} and \texttt{sw} access one word at a time.
- A pointer (used by \texttt{lw} and \texttt{sw}) is just a memory address, we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at runtime rather than compile-time.
- C Decisions are made using conditional statements within \texttt{if}, \texttt{while}, \texttt{do while}, \texttt{for}.
- MIPS Decision making instructions are the conditional branches: \texttt{beq} and \texttt{bne}.
- One can store and load (signed and unsigned) \texttt{bytes} as well as words with \texttt{lb}, \texttt{lbu}.
- Unsigned add/sub don’t signal overflow.
- Loops using \texttt{beq} and \texttt{bne}.
- New Instructions:
  \texttt{lw, sw, beq, bne, j, lb, sb, lbu, addu, addiu, subu, srl, sll} ... WOW
Inequalities in MIPS (1/4)

- Until now, we’ve only tested equalities (== and != in C). General programs need to test < and > as well.

- Introduce MIPS Inequality Instruction:
  - “Set on Less Than”
  - Syntax: `slt reg1,reg2,reg3`
  - Meaning: `reg1 = (reg2 < reg3);`

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

“set” means “change to 1”, “reset” means “change to 0”.

Same thing…
Inequalities in MIPS (2/4)

- How do we use this? Compile by hand:
  ```
  if (g < h) goto Less; #g:$s0, h:$s1
  ```

- Answer: compiled MIPS code...
  ```
  slt $t0,$s0,$s1    # $t0 = 1 if g<h
                   # $t0 = 0 if g>=h
  bne $t0,$0,Less   # goto Less
                   # if $t0!=0
  ```

- Register $0 always contains the value 0, so bne and beq often use it for comparison after an slt instruction.

- A slt ➔ bne pair means if(… < …) goto...
Inequalities in MIPS (3/4)

- Now we can implement $<$, but how do we implement $>$, $\leq$ and $\geq$?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement $\leq$ in one or more instructions using just $\text{slt}$ and branches?
  - What about $>?$
  - What about $\geq$?
Inequalities in MIPS (4/4)

- Lets compile this by hand:
  \[
  \text{if (} g \geq h \text{) goto Less; } #g: \$s0, h: \$s1
  \]

- Answer: compiled MIPS code…
  \[
  \text{slt } \$t0, \$s0, \$s1 \quad \# \quad \text{if } g < h
  \]
  \[
  \text{beq } \$t0, \$0, \text{Less} \quad \# \quad \text{goto less if } g \geq h
  \]

Two independent variations possible:

- Use \text{slt} \ $t0, \$s1, \$s0 \ instead of \text{slt} \ $t0, \$s0, \$s1
- Use \text{bne} instead of \text{beq}
Immediates in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in `for` loops

```c
if (g >= 1) goto Loop;
```

```mips
slti $t0,$s0,1  # $t0 = 1 if g < i
beq  $t0,$0,Loop  # goto Loop if g >= 1
```

An `slt` ➔ `beq` pair means `if(...) ≥ (...) goto...`
What about **unsigned** numbers?

- Also **unsigned** inequality instructions: `sltu, sltiu`
  - which sets result to 1 or 0 depending on unsigned comparisons

- What is value of $t0, t1$?
  - ($s0 = FFFF FFFA_{hex}, s1 = 0000 FFFA_{hex}$)
  - `slt $t0, $s0, $s1`
  - `sltu $t1, $s0, $s1`
MIPS Signed vs. Unsigned – diff meanings!

- MIPS terms Signed/Unsigned “overloaded”:
  - Do/Don't sign extend
    - (lb, lbu)
  - Do/Don't overflow
    - (add, addi, sub, mult, div)
    - (addu, addiu, subu, multu, divu)
  - Do signed/unsigned compare
    - (slt, slti/sltn, sltiu)
Peer Instruction

Loop: addi $s0, $s0, -1  # i = i - 1
       slti $t0, $s1, 2   # $t0 = (j < 2)
       beq $t0, $0, Loop  # goto Loop if $t0 == 0
       slt $t0, $s1, $s0  # $t0 = (j < i)
       bne $t0, $0, Loop  # goto Loop if $t0 != 0

(($s0 = i, $s1 = j))

What C code properly fills in the blank in loop below?

```c
do {i--;} while(____);
```
Administrivia

- Project 1 due Friday!
  - (ok, Saturday, but tell your brain it’s Friday!)
- Projector bulb is dying
  - I tried to adjust the color scheme accordingly. I called ETS. Should be fixed soon.
- HW 3 and 4 now online
- Midterm:
  - TWO WEEKS FROM TOMORROW!
    - Friday, July 16th. 9:30am-12:30pm. Room: 100 Lewis
- Other administrivia?
C functions

```c
main() {
    int i, j, k, m;
    ...
    i = mult(j, k); ...
    m = mult(i, i); ...
}

/* really dumb mult function */
int mult (int mcand, int mlier){
    int product = 0;
    while (mlier > 0) {
        product = product + mcand;
        mlier = mlier - 1; }
    return product;
}
```

What information must compiler/programmer keep track of?

What instructions can accomplish this?
Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.

- Register conventions:
  - Return address $ra$
  - Arguments $a0, a1, a2, a3$
  - Return value $v0, v1$
  - Local variables $s0, s1, \ldots, s7$

- The stack is also used; more later.
In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}

address (shown in decimal)

1000  add  $a0,$s0,$zero  # x = a
1004  add  $a1,$s1,$zero  # y = b
1008  addi $ra,$zero,1016  #$ra=1016
1012  j sum  #jump to sum
1016 ...

2000  sum:  add  $v0,$a0,$a1
2004  jr  $ra  # new instruction
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}

• Question: Why use `jr` here? Why not use `j`?

• Answer: `sum` might be called by many places, so we can’t return to a fixed place. The calling proc to `sum` must be able to say “return here” somehow.

2000 `sum:` add $v0,$a0,$a1
2004 `jr` $ra  # new instruction
Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (jal)

- Before:
  
  1008 addi $ra,$zero,1016 #$ra=1016
  1012 j sum #goto sum

- After:
  
  1008 jal sum # $ra=1012,goto sum

- Why have a jal?
  
  - Make the common case fast: function calls very common.
  - Don’t have to know where code is in memory with jal!
Instruction Support for Functions (5/6)

- Syntax for `jal` (jump and link) is same as for `j` (jump):
  
  ```
  jal label
  ```

- `jal` should really be called `laj` for “link and jump”:
  - Step 1 (link): Save address of next instruction into `$ra`
   - Why next instruction? Why not current one?
  - Step 2 (jump): Jump to the given label
Instruction Support for Functions (6/6)

- Syntax for `jr` (jump register):
  ```
  jr register
  ```

- Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to.

- Very useful for function calls:
  - `jal` stores return address in register ($ra)
  - `jr $ra` jumps back to that address
Nested Procedures (1/2)

```c
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- Something called `sumSquare`, now `sumSquare` is calling `mult`.
- So there’s a value in $ra$ that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.
- Need to save `sumSquare` return address before call to `mult`.
Nested Procedures (2/2)

- In general, may need to save some other info in addition to $ra.
- When a C program is run, there are 3 important memory areas allocated:
  - **Static**: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
  - **Heap**: Variables declared dynamically via `malloc`
  - **Stack**: Space to be used by procedure during execution; this is where we can save register values
C memory Allocation review

Address-> ∞

Stack

Heap

Static

Code

Space for saved procedure information

Explicitly created space, i.e., `malloc()`

Variables declared once per program; e.g., globals

Program
Using the Stack (1/2)

- So we have a register $sp$ which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```c
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```
Using the Stack (2/2)

Hand-compile

```c
int sumSquare(int x, int y) {
    sumSquare: return mult(x,x)+ y;
}
```

```
addi $sp,$sp,-8  # space on stack
sw $ra, 4($sp)  # save ret addr
sw $a1, 0($sp)  # save y
add $a1,$a0,$zero  # mult(x,x)
jal mult  # call mult
lw $a1, 0($sp)  # restore y
add $v0,$v0,$a1  # mult()+y
lw $ra, 4($sp)  # get ret addr
addi $sp,$sp,8  # restore stack
jr $ra
```

```
mult: ...
```

“push”

```
addi $sp,$sp,-8
sw $ra, 4($sp)
sw $a1, 0($sp)
add $a1,$a0,$zero
jal mult
```

“pop”

```
lw $ra, 4($sp)
lw $a1, 0($sp)
add $v0,$v0,$a1
lw $ra, 4($sp)
addi $sp,$sp,8
jr $ra
```

Steps for Making a Procedure Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. jal call
4. Restore values from stack.
Rules for Procedures

- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0`, `$a1`, `$a2` and `$a3`
- Return value is always in `$v0` (and if necessary in `$v1`)
- Must follow register conventions

So what are they? NEXT TIME
Basic Structure of a Function

Prologue

entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)  # save $ra
save other regs if need be

Body  . . . (call other functions ...)

Epilogue

restore other regs if need be
lw $ra, framesize-4($sp)  # restore $ra
addi $sp,$sp, framesize
jr $ra
### MIPS Registers

<table>
<thead>
<tr>
<th>Category</th>
<th>Register Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>The constant 0</td>
<td>$0$</td>
</tr>
<tr>
<td>Reserved for Assembler</td>
<td>$1$</td>
</tr>
<tr>
<td>Return Values</td>
<td>$2-3$</td>
</tr>
<tr>
<td>Arguments</td>
<td>$4-7$</td>
</tr>
<tr>
<td>Temporary</td>
<td>$8-15$</td>
</tr>
<tr>
<td>Saved</td>
<td>$16-23$</td>
</tr>
<tr>
<td>More Temporary</td>
<td>$24-25$</td>
</tr>
<tr>
<td>Used by Kernel</td>
<td>$26-27$</td>
</tr>
<tr>
<td>Global Pointer</td>
<td>$28$</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>$29$</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>$30$</td>
</tr>
<tr>
<td>Return Address</td>
<td>$31$</td>
</tr>
</tbody>
</table>

- **Already discussed**
- **New**
- **Not yet**

(From MIPS green sheet) Use names for registers -- code is clearer!
Other Registers

- **$at**: may be used by the assembler at any time; unsafe to use
- **$k0–$k1**: may be used by the OS at any time; unsafe to use
- **$gp, $fp**: don’t worry about them
- Note: Feel free to read up on $gp and $fp in Appendix A, but you can write perfectly good MIPS code without them.
When translating this to MIPS...

1) **We COULD** copy $a0 to $a1 (\& then not store $a0 or $a1 on the stack) to store \( n \) across recursive calls.

2) **We MUST** save $a0 on the stack since it gets changed.

3) **We MUST** save $ra on the stack since we need to know where to return to...

```c
int fact(int n)
{
    if(n == 0) return 1; else return(n*fact(n-1));
}
```
“And in Conclusion…”

- In order to help the conditional branches make decisions concerning inequalities, we introduce a single instruction: “Set on Less Than” called `slt`, `slti`, `sltu`, `sltiu`
- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…
  - Arithmetic: `add`, `addi`, `sub`, `addu`, `addiu`, `subu`
  - Memory: `lw`, `sw`, `lb`, `sb`
  - Decision: `beq`, `bne`, `slt`, `slti`, `sltu`, `sltiu`
  - Unconditional Branches (Jumps): `j`, `jal`, `jr`
- Registers we know so far
  - All of them!
“And in Conclusion to the conclusion…”

We are $\frac{1}{4}$ of the way done!
Example: The C Switch Statement (1/3)

- Choose among four alternatives depending on whether \( k \) has the value 0, 1, 2 or 3. Compile this C code:

```c
switch (k) {
    case 0: f=i+j; break; /* k=0 */
    case 1: f=g+h; break; /* k=1 */
    case 2: f=g-h; break; /* k=2 */
    case 3: f=i-j; break; /* k=3 */
}
```
This is complicated, so simplify.

Rewrite it as a chain of if-else statements, which we already know how to compile:

\[
\begin{align*}
&\text{if}(k==0) \ f=i+j; \\
&\quad \text{else if}(k==1) \ f=g+h; \\
&\quad \quad \text{else if}(k==2) \ f=g-h; \\
&\quad \quad \quad \text{else if}(k==3) \ f=i-j;
\end{align*}
\]

Use this mapping:

\[
\begin{align*}
f&:\$s0, \ g:\$s1, \ h:\$s2, \\
i&:\$s3, \ j:\$s4, \ k:\$s5
\end{align*}
\]
Example: The C Switch Statement (3/3)

- Final compiled MIPS code:

```
bne $s5,$0,L1    # branch k!=0
add $s0,$s3,$s4 #k==0 so f=i+j
j Exit          # end of case so Exit

L1: addi $t0,$s5,-1  # $t0=k-1
bne $t0,$0,L2    # branch k!=1
add $s0,$s1,$s2 #k==1 so f=g+h
j Exit          # end of case so Exit

L2: addi $t0,$s5,-2  # $t0=k-2
bne $t0,$0,L3    # branch k!=2
sub $s0,$s1,$s2 #k==2 so f=g-h
j Exit          # end of case so Exit

L3: addi $t0,$s5,-3  # $t0=k-3
bne $t0,$0,Exit  # branch k!=3
sub $s0,$s3,$s4 # k==3 so f=i-j
Exit:
```