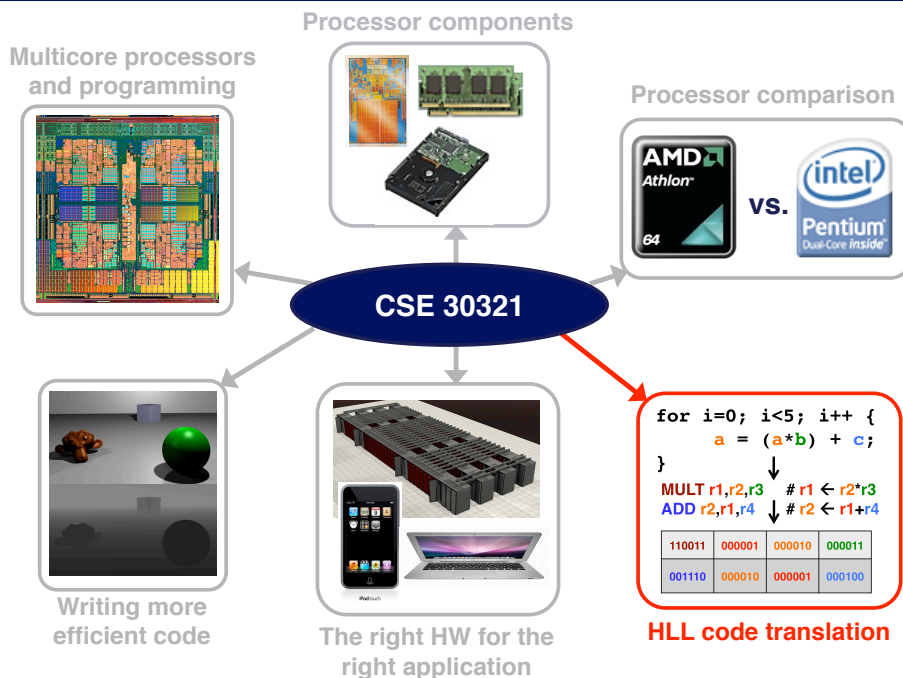


Readings

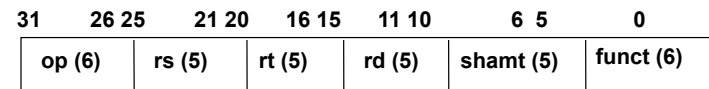
- Readings
 - H&P: Chapter 2.8
 - For extra examples, see Chapters 2.12 and 2.13

Lecture 07-08 MIPS and Programs and Procedures

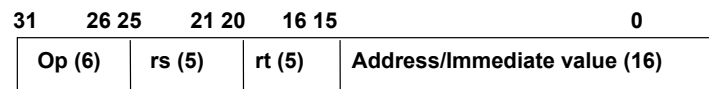


Summary of MIPS Instruction Formats

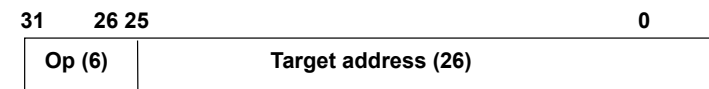
- All MIPS instructions are 32 bits (4 bytes) long.
- R-type:

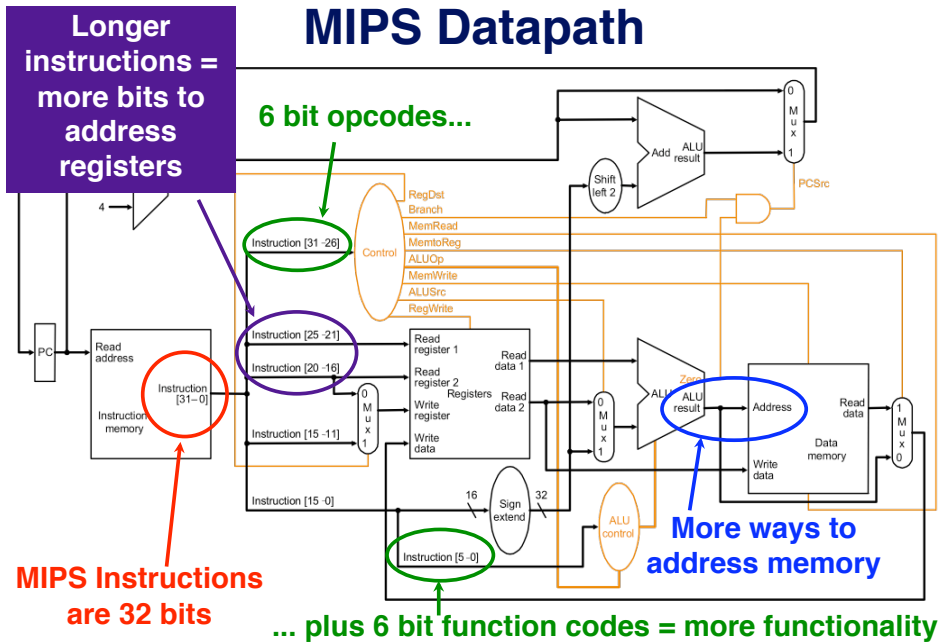


- I-Type:



- J-type





Practical Procedures

Have already seen that you don't necessarily make N copies of for loop body

Thus:

```
for (i=0; i<N; i++) {
    a = b + c;
    d = a + e;
    f = d + i;
}
```

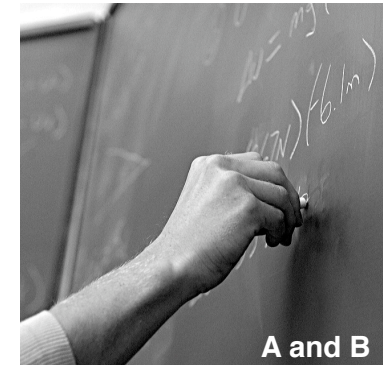
Might look like this:

```
# N = $2, i = $3
subi $2, $2, 1      # N = N - 1
loop: add $4, $5, $6 # a = b + c
      add $7, $4, $8 # d = a + e
      add $9, $7, $10 # f = d + i
      addi $3, $3, 1 # i = i + 1
      sub $11, $2, $3 # $11 = $3 - $2
      bneq $11, $0, loop # if $11 != 0, loop
```

You wouldn't make multiple copies of a machine instruction function either...

Three Discussion Items

- J type instruction encodings
- Endian-ness
- Common constant values



Practical Procedures

For example:

```
int main(void) {
    int i;
    int j;

    j = power(i, 7);
}

int power(int i, int n) {
    int j, k;
    for (j=0; j<n; j++)
        k = i*i;
    return k;
}
```

Might look like this:

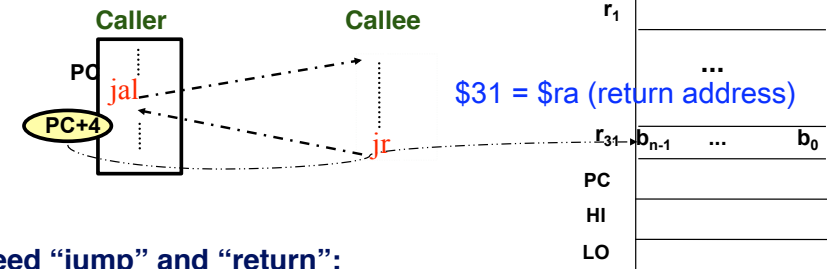
```
i = $6 # i in an arg reg.
addi $5, $0, 7 # arg reg. = 7
j power
call:
....
power: add $3, $0, $0
      subi $5, $5, 1
loop: mult $6, $6, $6
      addi $3, $3, 1
      sub $11, $5, $3
      bneq $11, $0, loop
      add $2, $6, $0 # data in ret. reg.
      j call
```

Advantage: Much greater code density. (especially valuable for library routines, etc.)

Procedure calls are so common that there's significant architectural support.

MIPS Procedure Handling

□ The big picture:



□ Need “jump” and “return”:

- **jal ProcAddr** # issued in the caller
 - jumps to ProcAddr
 - save the return instruction address in \$31
 - PC = JumpAddr, RF[31]=PC+4;
- **jr \$31 (\$ra)** # last instruction in the callee
 - jump back to the caller procedure
 - PC = RF[31]

MIPS Registers

(and the “conventions” associated with them)

Name	R#	Usage	Preserved on Call
\$zero	0	The constant value 0	n.a.
\$at	1	Reserved for assembler	n.a.
\$v0-\$v1	2-3	Values for results & expr. eval.	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$k0-\$k1	26-27	Reserved for use by OS	n.a.
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

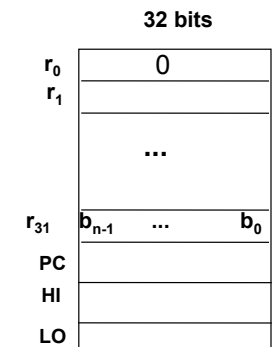
MIPS Procedure Handling (cont.)

□ What about passing parameters and return values?

- registers \$4 - \$7 (\$a0-\$a3) are used to pass first 4 parameters
- returned values are in \$2 and \$3 (\$v0-\$v1)

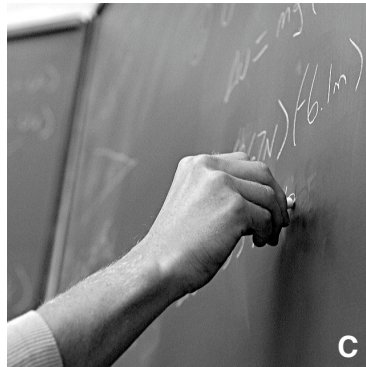
□ 32x32-bit GPRs (General purpose registers)

- \$0 = \$zero
- \$2 - \$3 = \$v0 - \$v1 (return values)
- \$4 - \$7 = \$a0 - \$a3 (arguments)
- \$8 - \$15 = \$t0 - \$t7 (temporaries)
- \$16 - \$23 = \$s0 - \$s7 (saved)
- \$24 - \$25 = \$t8 - \$t9 (more temporaries)
- \$31 = \$ra (return address)



Take away: HW support for SW tasks.

Swap Procedure Example



More complex cases

- Register contents across procedure calls are designated as either **caller or callee saved**
- MIPS register conventions: (although could make caller/callee do all)
 - $\$t^*$, $\$v^*$, $\$a^*$: not preserved across call
 - caller saves them if required
 - $\$s^*$, $\$ra$: preserved across call
 - callee saves them if required

What if ... ?

- More complex procedure calls
 - What if you have more than 4 arguments?
 - What if your procedure requires more registers than available?
 - What about nested procedure calls?
 - What happens to $\$ra$ if proc1 calls proc 2 which calls proc3,...

Recall...

(MIPS registers and associated “conventions”)

Name	R#	Usage	Preserved on Call
$\$zero$	0	The constant value 0	n.a.
$\$at$	1	Reserved for assembler	n.a.
$\$v0-\$v1$	2-3	Values for results & expr. eval.	no
$\$a0-\$a3$	4-7	Arguments	no
$\$t0-\$t7$	8-15	Temporaries	no
$\$s0-\$s7$	16-23	Saved	yes
$\$t8-\$t9$	24-25	More temporaries	no
$\$k0-\$k1$	26-27	Reserved for use by OS	n.a.
$\$gp$	28	Global pointer	yes
$\$sp$	29	Stack pointer	yes
$\$fp$	30	Frame pointer	yes
$\$ra$	31	Return address	yes

Where is all this stuff saved to?

Stack

- A dedicated area of memory
- First-In-Last-Out (FILO)
- Used to
 - Hold values passed to a procedure as arguments
 - Save register contents when needed
 - Provide space for variables local to a procedure

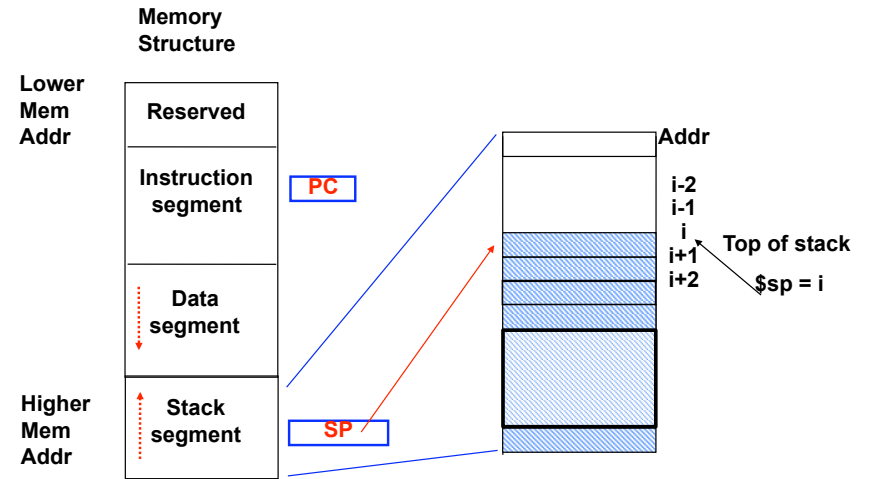
Stack operations

- push: place data on stack (*sw* in MIPS)
- pop: remove data from stack (*lw* in MIPS)

Stack pointer

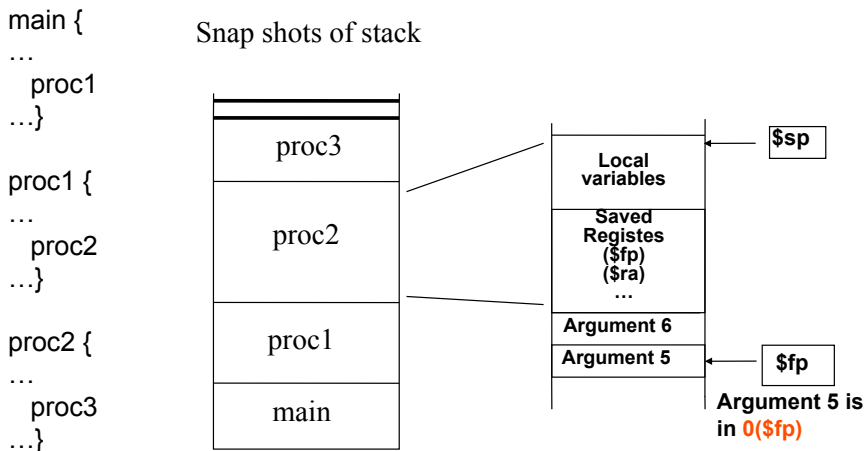
- Stores the address of the top of the stack
- **\$29 (\$sp)** in MIPS

Where is the stack located?



Call frames

- Each procedure is associated with a call frame
- Each frame has a frame pointer: **\$fp (\$30)**



Because $\$sp$ can change dynamically, often easier/intuitive to reference extra arguments via stable $\$fp$ – although can use $\$sp$ with a little extra math

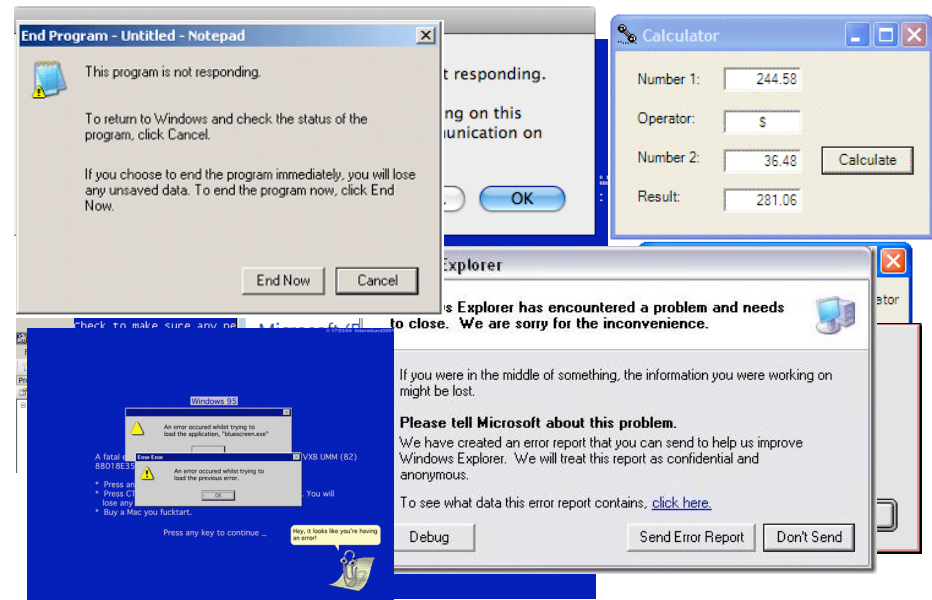
Procedure call essentials: Good Strategy

- Caller at call time
 - put arguments in $\$a0..\$a4$
 - save any caller-save temporaries
 - `jal ..., $ra`
 - Callee at entry
 - allocate all stack space
 - save $\$ra, \$fp + \$s0..\$s3$ if necessary
 - Callee at exit
 - restore $\$ra, \$fp + \$s0..\$s3$ if used
 - deallocate all stack space
 - put return value in $\$v0$
 - Caller after return
 - retrieve return value from $\$v0$
 - restore any caller-save temporaries
- do most work at callee entry/exit
- most of the work

Procedure call essentials

- **Summary**
 - **Caller saves registers**
 - (outside the agreed upon convention i.e. \$ax) at point of call
 - **Callee saves registers**
 - (per convention i.e. \$sx) at point of entry
 - **Callee restores saved registers, and re-adjusts stack before return**
 - **Caller restores saved registers, and re-adjusts stack before resuming from the call**

Why so strict?



Examples:

- Previous Exam Questions
- MIPS leaf procedure
- Nested function calls
 - **Stack pointers and frame pointers**
- Capstone example:
 - **Recursive Factorial!**

```
int fact(int n)
{
    if (n < 1)
        return (1);
    else
        return (n * fact(n-1));
}
```

