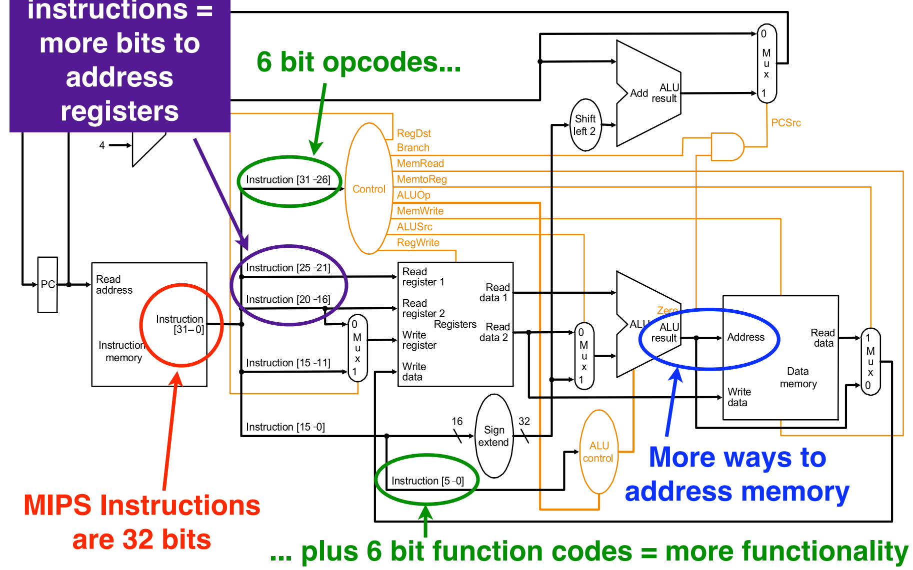
Lecture 08
Introduction to the MIPS ISA
+
Procedure Calls in MIPS

MIPS Datapath



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

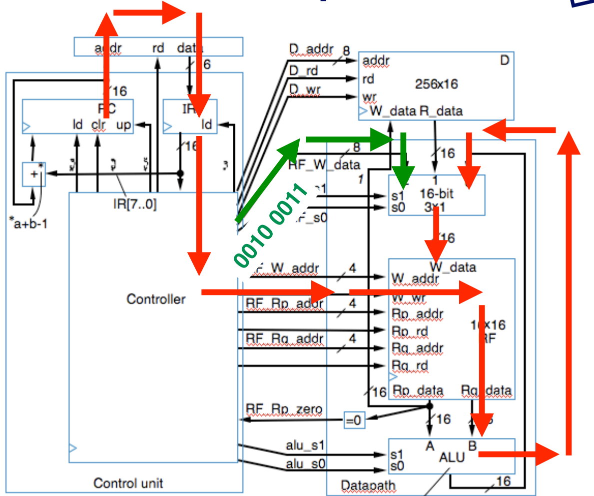
MIPS Instruction Types

- Instructions are characterized into basic types
- For each type 32 bits of instruction are interpreted differently
- 3 types of instructions in MIPS
  - R type
  - I type
  - J type
- In other words:
  - As seen with Add, instruction encoding broken down into X different fields
  - With MIPS, only 3 ways X # of bits arranged
    - Think about datapath: Why might this be good?

A quick look: more complex ISAs

Datapath

Path of Add from start to finish.



s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

Add: 0010 0001 0010 0011

Bits for Load C also "sent" 0010 0011

More types = more multiplexor inputs, signal routing

R-Type: Assembly and Machine Format

- R-type: All operands are in registers

Assembly: add $9, $7, $8 # add rd, rs, rt: RF[rd] = RF[rs]+RF[rt]

(add: op+func)

31 26	25 21	20 16	15 11	10 6	5 0
op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)

Machine:

B:	000000	00111	01000	01001	xxxxx	100000
D:	0	7	8	9	x	32

R-type Instructions

- All instructions have 3 operands
- All operands must be registers
- Operand order is fixed (destination first)
- Example:

C code: `A = B - C;`

(Assume that A, B, C are stored in registers s0, s1, s2.)

MIPS code: sub $s0, $s1, $s2

Machine code: 000000 10001 10010 10000 xxxxx 100010

- Other R-type instructions
  - addu, mult, and, or, sll, srl, …

I-Type Instructions

- I-type: One operand is an immediate value and others are in registers

Example: addi $s2, $s1, 128 # addi rt, rs, Imm
RF[18] = RF[17]+128

31 26	25 21	20 16	15 0
Op (6)	rs (5)	rt (5)	Address/Immediate value (16)

B:	001000	10001	10010	0000000010000000
D:	8	17	18	128

I-Type Instructions: Another Example

- I-type: One operand is an immediate value and others are in registers

Example: lw $s3, 32($t0) # RF[19] = DM[RF[8]+32]

31 26	25 21	20 16	15 0
Op (6)	rs (5)	rt (5)	Address/Immediate value (16)

B: 100011 01000 10011 0000000000100000
D: 35 8 19 32

How about load the next word in memory?

I-Type Instructions: Yet Another Example

- I-type: One operand is an immediate value and others are in registers

Example: Again:: bne $t0, $t1, Again
if (RF[8]!=RF[9]) PC=PC + 4 + Imm*4
else PC=PC+4

31 26	25 21	20 16	15 0
Op (6)	rs (5)	rt (5)	Address/Immediate value (16)

B: 00101 01000 01001 0000 0000 0001 0000
D: 5 8 9 16

PC-relative addressing

Byte addressability

- What “immediate values” are encoded in an I-type instruction (for example) are affected by the fact that MIPS data words are byte addressable
  - (Let’s look at Questions #1 and #2 on the board)

J-Type Instructions

- J-type: only one operand: the target address

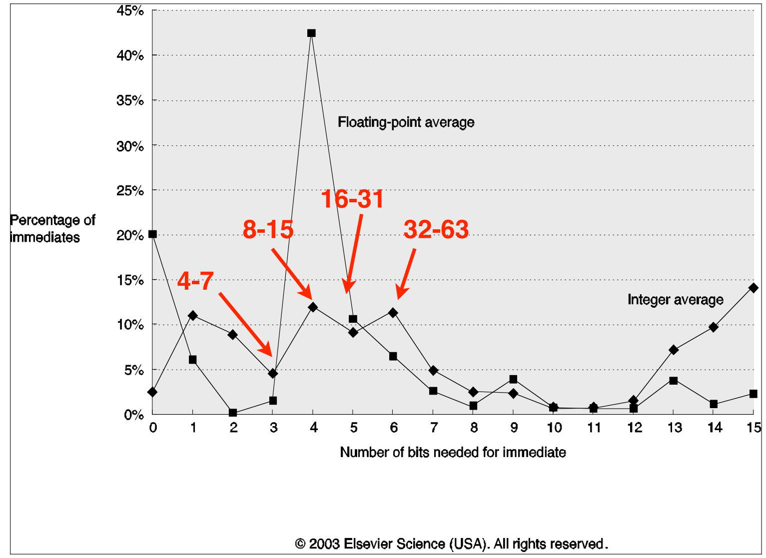
Example: j 3 # Goto addr. 3 x 4 (i.e. goto addr. 12)

31 26	25 0
Op (6)	Target address (26)

B: 000010 00000000000000000000000011
D: 2 3

In class examples

Size of Immediate Operand



Practical Procedures

Have already started to see that you don't 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...

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:

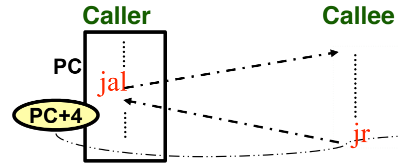
```
        i = $6               # i in an arg reg.

        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
```

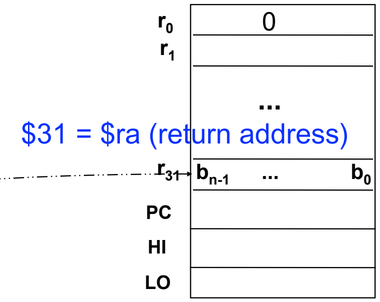
Advantage: Much greater code density. (especially valuable for library routines, etc.)

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

MIPS Procedure Handling

- The big picture:

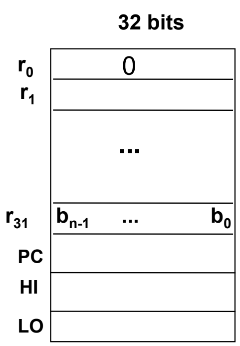




- Need "jump" and "return":
  - *jal ProcAddr* # issued in the caller
    - jumps to ProcAddr
    - save the return instruction address in $31
    - PC = JumpAddr, RF[31]=PC+4;
  - *jr $31* ($ra) # last instruction in the callee
    - jump back to the caller procedure
    - PC = RF[31]

MIPS 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)



In class example

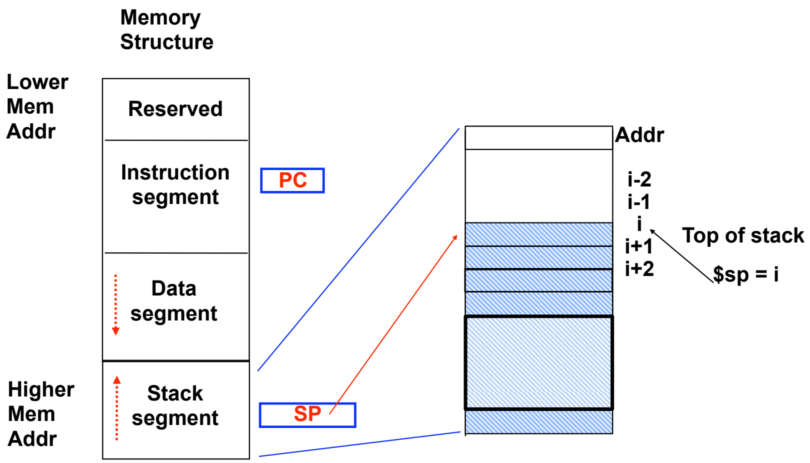
More complex cases

- Register contents across procedure calls are designated as either caller or callee saved
- MIPS register conventions:
  - $t*, $v*, $a*: not preserved across call
    - caller saves them if required
  - $s*, $ra, $fp: preserved across call
    - callee saves them if required
  - See P&H FIGURE 2.18 (p.88) for a detailed register usage convention
  - Save to where??
- More complex procedure calls
  - What if your 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,…

The stack comes to the rescue

- 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)

```
main {
…
  proc1
…}

proc1 {
…
  proc2
…}

proc2 {
…
  proc3
…}
```