Lecture 09
Procedure Calls in MIPS

MIPS Instruction Types

- R-type: All operands are in registers

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

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

- I-Type: 1 operand = immediate value, others 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)

- 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)

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

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;
}
```

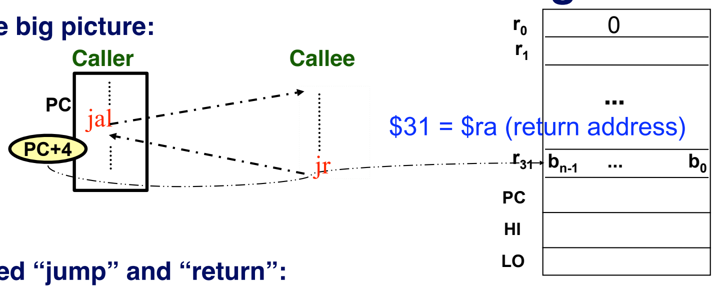
Might look like this:

```
        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.)

MIPS Procedure Handling

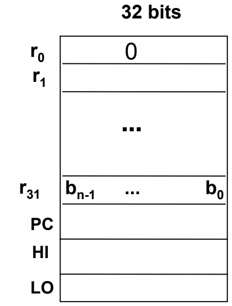
- The big picture:



- 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)



Review example

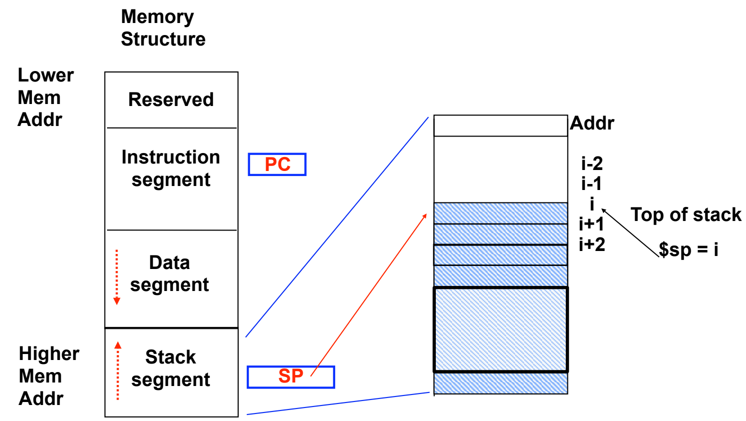
More complex cases

- 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?

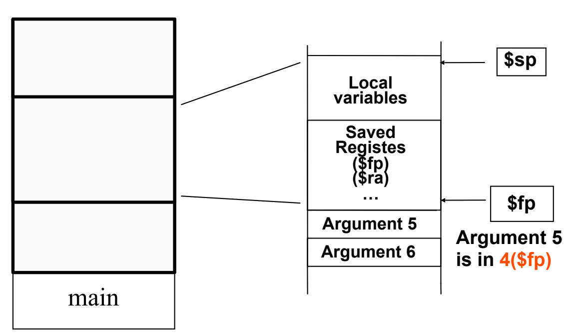


Call frames

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

```
main {
…
  proc1
…}

proc1 {
…
  proc2
…}

proc2 {
…
  proc3
…}
```

Snap shots of stack



Procedure call essentials (1): Caller/Callee Mechanics

- Four places

Who does what when?

```
foo()
{
    1. caller at call time
    bar(42);
    4. caller after return
}
```

```
bar(int a)
{
    2. callee at entry
    int temp = 3;
    ...
    return(temp + a);
    3. callee at exit
}
```

Procedure call essentials (2): Typical RISC machine (MIPS)

Nam	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

Procedure call essentials (3): Good Strategy

- Caller at call time
  - put arguments in $a0..$a4
  - save any caller-save temporaries
  - jalr ..., $ra
- Callee at entry
  - allocate all stack space
  - save $ra + $s0..$s3 if necessary
- Callee at exit
  - restore $ra + $s0..$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 (4)

- 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

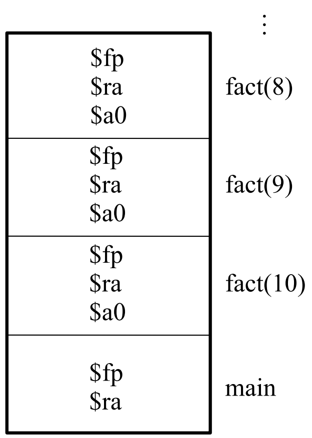
Example (Nested Function Calls)

Board discussion: stack and frame pointers

Example (As Complicated as It Gets)

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


Stack of Fact Recursion



- Assume
  - main calls fact at X1, $sp=Y0, $ra=X0
- Right after entering fact:
  - $ra = X1+4
  - $sp = Y0
- Right before calling fact again:
  - $ra = X1+4
  - $sp = Y0 - 32
- Right after returning from fact to main:
  - $sp = Y0
  - $ra = X1+4

MIPS ISA Summary

- Primarily supports 32 bit data (8, 16, and 64 bit also possible), byte addressable
- Multiple separate register spaces
  - 32 General Purpose Registers
  - Floating Point Registers
  - several dedicated registers for specific functions, e.g., PC, IR, Status, etc.
  - separate register sets for coprocessors
- Fixed-length, mostly horizontal instructions
- A load/store architecture: only load/store instructions can access memory
- Similar to other post-1980's architectures
  - (and pre-1980's architectures developed by Seymour Cray)

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as "RISC vs. CISC"
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- Examples: PowerPC and x86

PowerPC

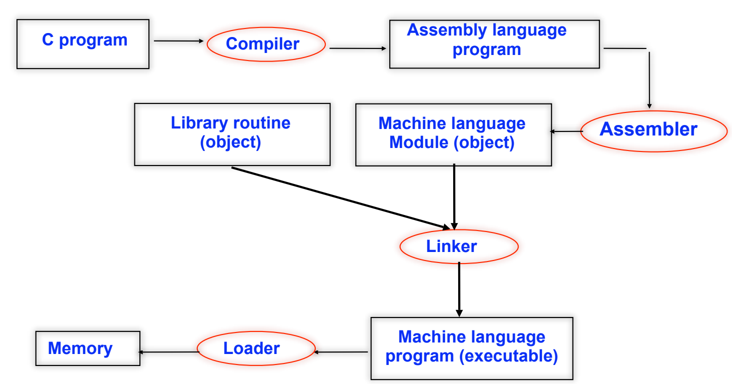
- Indexed addressing
  - example: `lw $t1,$a0+$s3 #$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load
    - (for marching through arrays)
  - example:
    - `lwu $t0,4($s3) #$t0=Memory[$s3+4]`
    - `$s3=$s3+4`
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register "`bc Loop`"
    decrement counter, if not 0 goto loop

80x86 (more "properly" IA32)

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    - e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

Translate a Program



The Role of Compilers

- Can you imagine writing 1000's or 10000's lines of assembly code?
- Compiler: convert HLL programs to assembly
- Goals of compiler design:
  - Correct code
  - Fast execution time (same as small instruction count?)
  - Fast compile time
  - Debugging support
  - …
- Architectural choices (ISA) affect the quality of the code that can be generated FOR the machine and the complexity of writing the compiler itself

Major Steps in a Compiler

Dependencies:	Steps	Function
Language dependent Machine independent	The Front End	HLL program => common intermediate representation (IR)
	↓ *IR form*	
Somewhat language dependent Largely machine independent	High-level optimizations	Optimize by changing IR structures
Less language dependent Somewhat machine dependent	Local/Global optimization	Optimize IR form within and beyond basic blocks
Less language dependent Somewhat machine dependent	Register allocation	Assign virtual registers to Actual registers
Highly machine dependent Language independent	Code generator	Select instructions and do machine-dependent optimizations

Typical Compiler Structure

Dependencies:		Function
Language dependent Machine independent	Front-end per language	Transform language to common intermediate form
	↓ *Intermediate representation*	
Somewhat language dependent Largely machine independent	High-level optimizations	For example, procedure inlining and loop transformations
Small language dependencies Machine dependencies slight (i.e. register counts/types)	Global optimizer	Including global and local optimizations + register allocation
Highly machine dependent Language independent	Code generator	Detailed instruction selection and machine-dependent optimizations (assembler next?)

More Details

- The front end:
  - IR form is similar to assembly, but assuming infinite number of registers
  - Check syntax, semantics, etc
- High-level optimization:
  - Often done on source w/output fed to later optimization passes
  - Examples: procedure inlining, loop unrolling, etc.
- Local Optimization:
  - Optimize code only within a straight-line code fragment (called a *basic block* by compiler folks)
  - Examples: common subexpression elimination, dead code elimination

More Details (cont'd)

- Global optimization:
  - Extend local optimizations across branches
  - introduce transformations aimed at optimizing loops
  - Must be conservative to guarantee correct code
- Register allocation:
  - Associate virtual registers to physical registers
  - Use life time analysis
  - Why want to maximize the use of registers?
- Code generation:
  - No separate assembler is needed
  - Attempt to take advantage of specific architectural knowledge

Many difficult optimization problems in compiler design!

Impact of Architecture on Compiler Design

- The compiler writer's manifesto:
  - Make the frequent cases fast and the rare cases correct!
- Guideline 1: Make things *REGULAR*:
  - Keep the components of ISA orthogonal (or independent) if at all possible => simplify code generation
- Guideline 2: Provide primitives, NOT solutions:
  - Don't target your architecture for a specific language. You're an architect, not a compiler writer!
- Guideline 3:
  - Simplify trade-offs among alternatives
- Guideline 4:
  - Provide instructions that bind quantities known at compile time as constants

Impact of Compiler and HLL on ISA

- Two important questions:
  - How are variables addressed and allocated?
  - How many registers are needed to allocate variables appropriately?
  - Data allocation in high-level languages
    - Stack
    - Global data area
    - Heap
  - Pointers make things hard!