CS61C
Functions, Procedures in C/Assembly Language

Lecture 4

January 29, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html

Review 1/1

- Constants so common have special version of arithmetic, registers
  - `addi`, `subi`; register `$zero` (always 0)
  - Principle: Making common case fast
- HLL decisions (if, case) and loops (while, for) use same assembly instructions
  - Conditional branches: `beq`, `bne` in MIPS
  - Unconditional branches: `j`, `jr` in MIPS
  - Relative test: `slt`, `slti` in MIPS
  - Case/Switch: either jump table + `jr` or simply chained if-else

Review 2/2

- MIPS assembly language instructions
  - Arithmetic: `add`, `sub`, `addi`, `subi`
  - Data transfer: `lw`, `sw`,
  - Relative test: `slt`, `slti`
  - Conditional branches: `beq`, `bne`
  - Unconditional branches: `j`, `jr`
- Operands
  - Registers (word = 32 bits): `$zero`; `$s0`, `$s1`, `. . .` ; `$t0`, `$t1`, `. . .` ;
  - Memory (8-bit byte address, 4 bytes/word): `Memory[0]`, `Memory[4]`, `Memory[8]`, `, . . . ,Memory[4294967292]`

Overview

- C functions (2 minutes)
- Bookkeeping for function call/return
- Instruction support for functions
- Nested function calls
- C memory allocation: static, heap, stack
- Administrivia,"Computers in the news"
- Resolving Registers Conflicts (6 min)
- Frame/Stack pointer (12)
- C/Assembly Examples (6 min)

C functions

```
main() {
 int i,j,k,m;

 i = mult(j,k); ... ;
 m = mult(i,i); ...

}

int mult (int mcand, int mlier){
 int product;

 product = 0;
 while (mlier > 0)  {
  product = product + mcand;
  mlier = mlier -1; }
 return product;

}
```

What information must compiler/programmer keep track of?

Function Call Bookkeeping

- Procedure address
- Return address
- Arguments
- Return value
- Local variables
- Registers (conflicts)

- Registers for functions
- `$ra`
- `$a0`, `$a1`, `$a2`, `$a3`
- `$v0`, `$v1`
- `$s0`, `$s1`, …, `$s7`;

Instruction Support for Functions?

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

MIPS

address

```
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
```

Why jr vs. j to return?

Instruction Support for Functions?

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

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum               #go to sum
```

- After:

```
1012 jal sum  # $ra=1016,go to sum
```

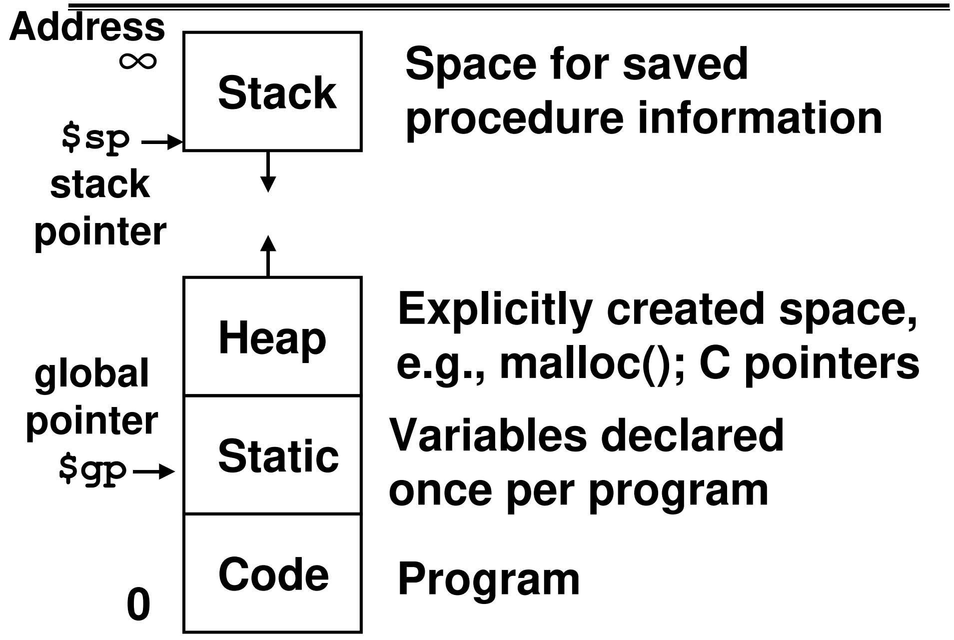
- Why `jal`? Make the common case fast

Nested Procedures

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

- **Need to save `sumSquare` return address before call `mult`**
  - Otherwise `jal mult` overwrites `$ra`
- **One word per procedure in memory ?**
  - e.g., `sw $ra, sumSquareRA($s3)`
- **Recursive procedures could overwrite saved area => need safe area per function invocation => stack**

C memory allocation seen by the Program



Compiling C `if` into MIPS: Summary

- **Compile by hand**

C

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

```
sumSquare:
           subi $sp,$sp,12 # space on stack
Prologue   sw $ra,$ 8($sp)  # save ret addr
           sw $a0,$ 0($sp)  # save x
           sw $a1,$ 4($sp)  # save y
Body       addi $a1,$a0,$zero # mult(x,x)
           jal mult         # call mult
           lw $ra,$ 8($sp)  # get ret addr
Epilogue   lw $a0,$ 0($sp)  # restore x
           lw $a1,$ 4($sp)  # restore y
           add $vo,$v0,$a1  # mult()+y
           addi $sp,$sp,12  # => stack space
           jr $ra
```

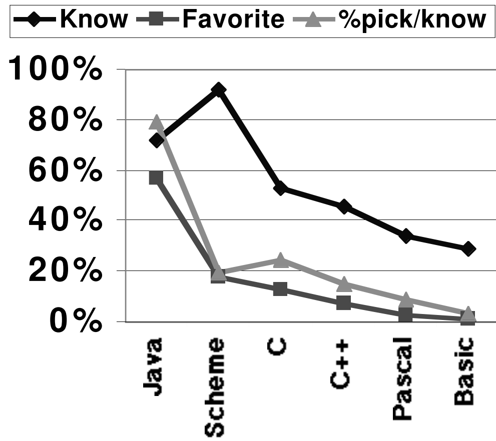

Exceeding limits of registers

- **Recall: assembly language has fixed number of operands, HLL doesn’t**
- **Local variables: $s0, ..., $s7**
  - What if more than 8 words of local variables?
- **Arguments; $a0, ..., $a3**
  - What if more than 4 words of arguments?
- **Place extra variables and extra arguments onto stack ($sp)**
- **Use temp registers and data transfers to access these variables**

Administrivia

- Readings: 3.6, Appendix A.6; next 3.4,3.8
- 1st project: C spelling checker `philspel` Due Wed. 2/3 7PM (do by yourself) `www-inst.EECS/~cs61c/handouts/proj1.pdf`
- Change from 1 week ago: team size < 3
- 1st homework today 7PM 2nd homework: Due Wed 2/3 7PM
  - Exercises 3.1, 3.2, 3.4, 3.6, 3.9; Which instruction set inside? Search WWW
- Apple iMAC
- Casio PalmPC
- Cisco Network Routers
- HP LaserJet 4000
- IBM PC
- Kodak DC260
- NASA Mars Rover
- Nintendo 64
- Sony Playstation
- Web TV set top box

Survey Results



- No Stairs? 10%
- Free energy? 5%
- Special? 7%

- 61A 94% UC, 3.2 GPA
- 61B 63% UC, 3.2 GPA
- 9C (S.P. C)? 10%
- 9C (S.P. C++)? 15%
- Printer? 60%@home, 20% elsewhere
- Print at night? 1/3 Before 8AM? 1/3

"Computers in the News"

- "Intel Alters Plan Said To Undermine PC Users' Privacy", 1st p., N.Y. Times 1/26/99
- Processor-specific IDs per chip accessed by SW and transmitted over the Internet
  - 96-bit unique serial number: 32 CPU type+64 ID
  - Idea: ID helps intellectual property protection, tying apps, information to a specific machine
- "Big Brother" inside? Boycott Intel!
  - No anonymity? Track 1 consumer over Internet?
- "The Intel Corporation yesterday reversed a plan to activate an identifying signature in its next generation of computer chips, bowing to protests that the technology would compromise the privacy of users."
  - Not removed; default now off on reboot

Function Call Bookkeeping: thus far

- Procedure address √
- Return address √
- Arguments √
- Return value √
- Local variables √
- Registers (conflicts)

Register Conflicts

- Procedure A calls Procedure B
  - A referred to as is "calling procedure" or "caller"
  - B referred to as is "called procedure" or "callee"
- Both A and B want to use the 32 registers, but must cooperate

Register Conflicts: 2 options (A calls B)

1) Called procedure/callee (B) leaves registers the way it found them (except \$ra); its B's job to save it before using it and then restore it: "callee saves"

- Since B only saves what it changes, more accurate is "callee saves (what it uses)"

2) B can use any register it wants; Calling procedure/caller A must save any register it wants to use after call of B: "caller saves"

- Since A knows what it needs after call, more accurate is "caller saves (if it wants to)"

MIPS Solution to Register Conflicts

- Divide registers into groups
  - Saved registers (\$s0-\$s7)
  - Temporary regs (\$t0-\$t9), Argument (\$a0-\$a3)
  - Some caller saved (if used) and some callee saved (if used)
- Caller (A) save/restore temporary (\$t0-\$t9) and argument (\$a0-\$a3) if needs them after the call; also \$ra => callee can use \$ti,\$ai,\$ra
- Callee (B) must save/restore saved registers (\$s0-\$s7) if it uses them => caller can \$si
  - Procedure that doesn't call another tries to use only temporary and argument registers

Memory Allocation

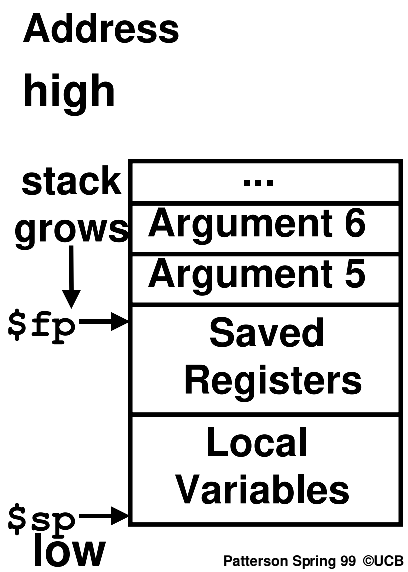
```
 ... sum(a,b);...
}
int sum(int x, int y) {
    return x+y;
}
```

CS61A name for?

a => x	Frame
b => y	

Memory Allocation

- C Procedure Call Frame
- Pass arguments (4 regs)
- Save caller-saved regs
- jal
- space on stack ($sp-n)
  $sp@last word of frame
- Save callee-saved regs
- set $fp ($sp+n-4)
  $fp@first word of frame



Frame Pointer Optional

- If allocate all stack space need at beginning of procedure (e.g., space for any arguments for any procedure might call),
- Then don't need to separate frame pointer to refer to variables; refer to every variable from $sp
- GCC uses frame pointer, MIPS compilers don't (get extra temporary register, less bookkeeping on procedure call)

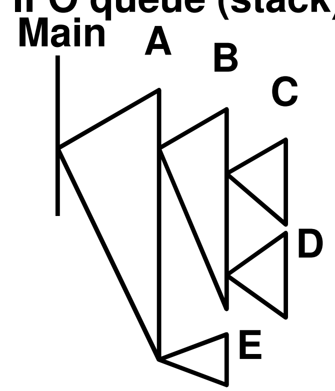
MIPS Register Summary

Registers	Total Regs
• $Zero	1
• (Return) Value registers ($v0,$v1)	3
• Argument registers ($a0-$a3)	7
• Return Address ($ra)	8
• Saved registers ($s0-$s7)	16
• Temporary registers ($t0-$t9)	26
• Global Pointer ($gp)	27
• Stack Pointer ($sp)	28
• Frame Pointer ($fp), or $t10	29

- 2 for OS ($k0, $k1), 1 for assembler ($at)

HLL and Frames

- C,C++, Java follow "Stack Discipline";
  - e.g., D cannot return to A
  - Frames can be adjacent in memory
  - Frames can be allocated, discarded as a FIFO queue (stack)



What about Scheme?

If there is time, Compile this MIPS code:

```
main() {
 int i,j,k,m; /* i-m:$s0-$s3 */

 i = mult(j,k); ... ;
 m = mult(i,i); ...

}

int mult (int mcand, int mlier){
 int product;

 product = 0;
 while (mlier > 0)  {
  product = product + mcand;
  mlier = mlier -1; }
 return product;

}
```


"And in Conclusion …" 1/2

- MIPS assembly language instructions
  - Arithmetic: `add,sub,addi, subi`
  - Data transfer: `lw, sw`
  - Relative test: `slt, slti`
  - Conditional branches: `beq, bne`
  - Unconditional branches: `j, jr, jal`
- Operands
  - Registers (word = 32 bits): `$zero`; `$v0, $v1, $a0-$a3, $s0-$s7, $t0-$t9, $gp, $sp, $fp, $ra`
  - Memory (8-bit byte address, 4 bytes/word): `Memory[0], Memory[4], Memory[8], , ... , Memory[4294967292]`

"And in Conclusion …" 2/2

- Functions, procedures one of main ways to give a program structure, reuse code
- `jr` required instruction; most add `jal` (or equivalent) to make common case fast
- Registers make programs fast, but make procedure/function call/return tricky
- MIPS SW convention divides registers into those calling procedure save/restore and those called procedure save/restore
  - Assigns registers to arguments, return address, return value, stack pointer
- Next Machine Representation: COD 3.4,3.8